Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jože Kulovic

Optimizacija avtomatskih skladiščnih sistemov

magistrsko delo

študijski program druge stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Uroš Lotrič

Ljubljana, 2016

Zahvaljujem se mentorju izr. prof. dr Urošu Lotriču za pomoč in strokovne nasvete pri izdelavi magistrskega dela.

Zahvaljujem se tudi staršem in ostalim, ki so mi stali ob strani vsa leta študija.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
AS/RS	Automated Storage and Retrieval Systems	Avtomatski skladiščni sistem
FIFO	First In First Out	Prvi noter in prvi ven
NP	Non-deterministic polynomial	Nedeterministično polinomen
TSP	Traveling Salesman Problem	Problem Trgovskega Potnika
V/I	Input/Output	Vhod/Izhod
WT	Warehouse Task	Skladiščna naloga - opravilo
WMS	Warehouse Management System	Sistem za upravljanje skladišča

Povzetek

Naslov: Optimizacija avtomatskih skladiščnih sistemov

V magistrskem delu smo razvili metode za optimizacijo razporeditve opravil skladiščnih sistemov v realnem času. Optimizacijo smo opravljali na podlagi obratujočega skladišča, iz katerega smo dobili vse specifikacije, tako da smo pri sami optimizaciji upoštevali tudi fizikalne lastnosti skladišča in naprav v njem. Skladiščna naprava za premikanje transportno skladiščnih enot izbranega skladiščnega sistema ima možnost sočasnega premikanja dveh transportno skladiščnih enot, kar nam daje veliko možnosti za optimizacijo. Za optimizacijo razporeditve opravil smo uporabili Dijkstrov algoritem, Bellman-Fordov algoritem, genetski algoritem, optimizacijo s kolonijami mravelj, algoritem A* in algoritem soseščine. Najboljše rezultate smo dosegli z algoritmom soseščine, ki temelji na odločitvenem pristopu. Optimizacijo razporeditve petstotih opravil je opravil v manj kot eni sekundi, pri tem pa smo dosegli 20 % pohitritev v primeru časovne optimizacije. V primeru energijske optimizacije pa smo porabo energije optimizirali za kar 40 %.

Ključne besede: avtomatizirano skladišče, dodeljevanje prostora, optimizacija, genetski algoritem, hevristični algoritem, primerjava algoritmov, računalništvo in informatika, optimizacija.

Abstract

Title: Optimisation of automated warehouse systems

In the master thesis we developed methods for optimizing tasks assignment in high bay warehouse system. Optimization was performed based on a real warehouse system from which we acquired all specifications. In our optimization we took into account all physical properties of the warehouse from the specification. Warehouse device for moving transport storage units in the selected warehouse can move two transport units at the same time, giving us a lot of optimization options. To optimize tasks assignment we used Dijkstra algorithm, Bellman-Ford algorithm, genetic algorithm, ant colony optimization algorithm, algorithm A* and neighborhood algorithm. We achieved the best result using neighborhood algorithm based on decision-making approach. Optimization of five hundred tasks was done in less than one second, and with it we increased task execution speed by 20 % and optimized energy consumption by 40 %.

Keywords: automated warehouse, warehouse storage allocation, optimization, genetic algorithm, heuristic algorithm, algorithm comparison, computer science, optimization.

Poglavje 1

Uvod

Skladišča so pomemben element prodajne in dobavne verige podjetij. Zaradi zahtev po večji učinkovitosti prodajnih in dobavnih verig so skladišča vedno bolj avtomatizirana in integrirana z računalniško tehnologijo za njihovo upravljanje in vodenje [1]. Opremljena so z napravami za skladiščenje in transport produktov. Produkte je potrebno na začetku pripeljati v skladišče, nato skladiščiti in na koncu odpeljati. V poslovnih procesih mnogih podjetij igra skladiščenje pomembno vlogo, zato je optimizacija skladiščnih postopkov postala predmet mnogih raziskav [2]. Optimizacija skladišč je kompleksno opravilo, ki je sestavljeno iz več NP-težkih problemov. Največ pozornosti posvečajo sistemom za dodeljevanje lokacij produktom, gibanju naprav v skladišču in razporejanju opravil med naprave [3].

V našem delu smo se osredotočili na razporejanje skladiščnih opravil skladiščni napravi v realnem času. Optimizirati želimo čas, ki je potreben za izvedbo opravil, ter energijo, ki se pri tem porabi za premikanje transportno skladiščnih enot. Za natančne izračune časa in energije bomo upoštevali fizikalne lastnosti zadanega skladiščnega sistema, ki obratuje v Turčiji. Zadani skladiščni sistem ima posebno skladiščno napravo za prenos transportno skladiščnih enot, ki zmore prenašati dve transportno skladiščni enoti hkrati. Naš cilj je tako ugotoviti, koliko učinkovitejši je lahko skladiščni sistem z

napravo, ki premore prenos dveh transportno skladiščnih enot, v primerjavi s skladiščnim sistemom, ki ima napravo, ki lahko prenaša le eno transportno skladiščno enoto. Pričakujemo, da bomo pri skladiščnem sistemu z napravo, ki premore prenos dveh transportno skladiščnih enot, dosegli veliko boljše optimizacijske rezultate kot pa pri skladiščnem sistemu, ki uporablja napravo z le enim mestom za transportno skladiščne enote. Pri optimizaciji energije, ki se porabi med prenašanjem transportno skladiščnih enot, pa želimo med sabo primerjati različne načine izrabe odvečne energije med pogoni skladiščne naprave in zopet ugotoviti njihovo učinkovitost.

Magistrsko delo smo razdelili na 6 poglavij. V poglavju 2 smo na splošno opisali skladiščne sisteme. Pri opisovanju smo se osredotočili na avtomatske skladiščne sisteme ter možne optimizacije v njih. V poglavju 3 smo opisali algoritme, ki smo jih uporabili pri optimizaciji razporeditve opravil. Algoritmi, ki smo jih uporabili pri optimizaciji, so bili: Dijkstrov algoritem, Bellman-Fordov algoritem, genetski algoritem, optimizacija s kolonijami mravelj, algoritem A* in algoritem soseščine. Po opisu uporabljenih algoritmov smo v poglavju 4 opisali skladiščni sistem, po katerem smo uporabili fizikalne parametre postavitve skladišča ter premikanja naprav v njem. V tem poglavju smo tudi opisali enačbe za izračun časov potrebnih za izvedbo premikov in porabo energije skladiščnih naprav. Nato smo v poglavju 5 opisali implementacijo ter parametre posameznih algoritmov, ki smo jih pri optimizaciji uporabili. Dobljene optimizacijske rezultate smo opisali v poglavju 6, kjer smo med seboj primerjali posamezne metode optimizacije. Poleg primerjave metod pa smo izvedli še primerjavo skladiščnega sistema pri različnih konfiguracijah. Delo smo nato zaključili s sklepom v poglavju 7.

Poglavje 2

Skladiščni sistemi

Skladiščni sistemi služijo shranjevanju produktov, ki jih proizvedejo tovarne, imajo trgovine zaradi svoje zaloge, hranijo uvozniki, izvozniki, podjetja ter druge dejavnosti.

Prva skladišča so se pojavila v 19. stoletju v času industrijske revolucije. Njihova najpomembnejša naloga je bila hranjenje produktov za vzdrževanje zaloge. S časom so skladišča pridobivala nove naloge in obveznosti; danes poleg osnovne naloge shranjevanja produktov skladiščni sistemi lahko služijo tudi za:

- *Vodenje inventure*: skladišča vodijo informacije o vseh produktih, ki so prišla v skladišče, in njihovih natančnih lokacijah v samem skladišču.
- *Zaščito produktov*: znotraj skladišča so vsi produkti običajno zaščiteni pred izgubo ter morebitno škodo zaradi vremenskih pojavov (dež, prah, veter, sneg, ...).
- *Prevzem odgovornosti*: ko pride produkt v skladišče, skladišče oziroma njegov imetnik postane odgovoren za produkt in vse, kar se bo z njim zgodilo, tako da se produkt iz skladišča vrne v enakem stanju, kot je vanj prišel.
- *Obdelave produktov*: nekatera skladišča lahko produkte dodatno obdelujejo po tem, ko so prišli v skladišče, na primer sušenje, zorenje lesa.

Ta skladišča so običajno velike zgradbe, ki vsebujejo viličarje, dvigala ter druge naprave za ravnanje s produkti. Avtomatizacija skladiščnih sistemov je v vsakdanjem življenju čedalje bolj pogosta. Razlogi zanjo so tako ekonomični kot praktični. Cene zemljišč so namreč zelo visoke, zaradi česar postanejo skladišča čedalje višja. Visoka skladišča pa so težavna za ročno upravljanje, zato se jih vedno bolj upravlja z avtomatskimi skladiščnimi sistemi. Poleg izboljšanega upravljanja skladišča so prednosti avtomatskih skladiščnih sistemov pred ročnim še [4]:

- *Manjše število kontaktov s produktom.* Pri ročnem shranjevanju ter pobiranju lahko pri transportu produktov skozi skladišče pride do napak, poškodb ali izgube produktov. Z avtomatskim skladiščnim sistemom se takšne napake odpravijo.

- *Optimalnejša mesta produktov.* Produkti so hranjeni po sistemski logiki, ki je vedno enaka; tako se prepreči, da bi se kakšen produkt izgubil oziroma založil. Pri ročnem shranjevanju bi uporabnik produkt običajno shranil na mesto, ki bi mu bilo bolj intuitivno.

- *Zanesljivejše delovanje.* Avtomatski skladiščni sistemi delujejo ves čas enako zanesljivo.

- *Večja učinkovitost.* Prepustnost je konstantna in v primerjavi z ročnimi skladiščnimi sistemi večja, ravno tako avtomatskim skladiščnim napravam višina ne predstavlja dodatnih težav – tako so višja skladišča še vedno enako učinkovita.

- *Večja kapaciteta skladišča.* Avtomatski skladiščni sistemi pri shranjevanju/pobiranju dosegajo večje natančnosti, zato so lahko predalniki nižji ter bolj stisnjeni. Posledično lahko na enako velikem območju povečamo kapaciteto skladišča.

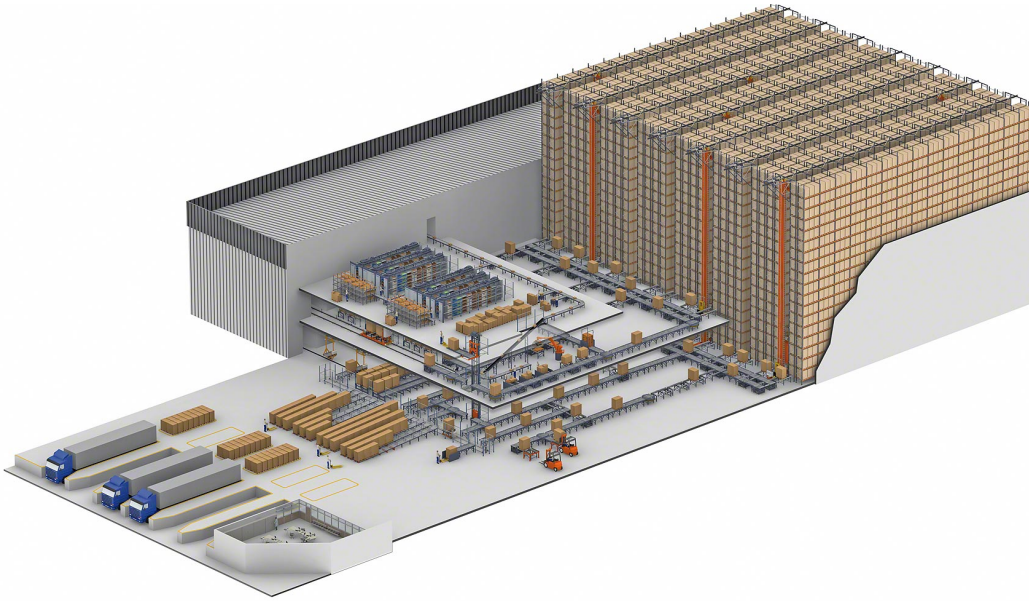
- *Neposredna povezava z informacijskem sistemom, ki vodi celoten sistem.* Z direktno povezavo z informacijskem sistemom se ves čas ve za vse produkte, kje se nahajajo znotraj skladišča.

2.1 Proces skladiščenja

Proces skladiščenja je del poslovne strategije posameznega podjetja. Učinkovita izvedba pripomore k učinkovitejšemu skladiščnemu sistemu, ki ima veliko prepustnost. Osnovni proces skladiščnega sistema od sprejema pa do pošiljanja produkta je sestavljen iz petih večjih aktivnosti:

1. sprejem (ang. *receiving*),
2. skladiščenje (ang. *storage*),
3. pobiranje (ang. *picking*),
4. pakiranje (ang. *packing*),
5. pošiljanje (ang. *shiping*).

Shema avtomatskega skladiščnega sistema je prikazana na sliki 2.1. Na shemi so tako prikazani: sprejemni/odpremni prostor, skladiščni prostor ter pisarna.



Slika 2.1: Shema avtomatskega skladiščnega sistema [5]

2.1.1 Sprejem

Sprejem je prva aktivnost v skladiščnem sistemu, s katero se celotni proces sistema začne. Prične se s prihodom tovornjaka, ladje ali druge tovorne naprave, ki ima naložene produkte, ki jih želi shraniti v skladišče. Pri sprejemu produktov v skladišče mora biti navzoča pristojna oseba, ki obravnava produkte. Produkte morajo pred vnosom v skladišče identificirati in prešteti, preveriti njihovo kakovost in stanje ter čas prihoda v primerjavi s pričakovanim in preveriti, ali imajo zanj ustrezne pogoje. Glede na politiko podjetja se lahko dodatno preveri še ostale atribute produktov v odvisnosti od njihove občutljivosti. Ko je posamezen produkt ustrezno obravnavan in je vse v skladu z internimi pravili podjetja, se produkt vnese v zalogo ter odda v sprejemni prostor, iz katerega se bo nato pomaknil v skladišče na ustrezno mesto.

2.1.2 Skladiščenje

Druga aktivnost skladiščnega procesa je skladiščenje produktov. Cilj aktivnosti je prenos produktov iz sprejemnega prostora na ustrezno mesto znotraj skladišča, za pridobivanje produktov iz skladišča pa bo poskrbela aktivnost pobiranje. Mesto znotraj skladišča, na katerega se bodo produkti shranili, je odvisno predvsem od politike podjetja. Nekateri načini določanja skladiščnih mest produktom znotraj skladišča so:

- naključni,
- veliki produkti na spodnjih skladiščnih mestih, majhni na zgornjih,
- produkte, ki bodo šli skupaj iz skladišča, hrani skupaj (v primeru, da sistem lahko napove, kateri produkti bodo šli skupaj iz skladišča),
- druge napredne metode za določanje skladiščnih mest produktom, s katerimi dosežejo boljšo učinkovitost skladiščnega sistema.

2.1.3 Pobiranje

Proces pobiranje se prične, ko dobimo določeno naročilo (zahtevo) in je potrebno izbrane produkte iz skladišča privesti na izhodno mesto skladišča. S prihodom naročila v skladišče je v primeru več enakih produktov potrebno najprej določiti, katere produkte s katerih skladiščnih mest se bo privedlo iz skladišča. Pri tem se lahko izbira po metodi FIFO ali pa po kakšni drugi metodi, ki jo določa politika podjetja. Ko so skladiščna mesta s produkti iz naročila natančno določena, je potrebno te samo še pobrati. Pobiranje produktov iz skladišča je relativno enostavna operacija, vendar je lahko časovno zelo zahtevna. V primeru, da so produkti, ki jih želimo vzeti iz skladišča, razpršeni po celotnem skladišču, je zelo pomemben vrstni red, po katerem se bomo lotili pobiranja produktov. Pobiranje produktov se lahko izvaja na dva načina. Pri prvem se lahko iz skladišča vzame celotna paleta, pri drugem pa se s palete vzame le nekaj produktov, paleta pa se vrne v skladišče.

2.1.4 Pakiranje

Pakiranje je aktivnost, pri kateri se produkti pripravijo za pošiljanje stranki. V primeru, da je bilo v naročilu več produktov in so bili ti iz skladiščnega sistema pobrani posamezno, se v tem koraku dodatno preveri, ali so vsi potrebni produkti izbrani ter ali so njihove količine pravilne. Produkti so nato glede na njihove specifikacijeustrezno zapakirani v škatle/zaboje in označeni.

2.1.5 Pošiljanje

Potem ko so paketi pripravljeni za razpošiljanje, ta aktivnost poskrbi, da se paketi ustrezno odpremijo. Pakete se običajno razpošilja s tovornjakom, ladjo ali drugo tovorno napravo, na katero jih lahko naložimo. Pri nalaganju paketov se običajno vsi paketi tehtajo ter zanje vodi evidenca. Za pošiljanje se lahko najame drugo podjetje, ki se ukvarja samo z razpošiljanjem paketov. Takšna podjetja običajno nudijo tudi zavarovanje v primeru, če bi se s paketom na poti kaj zgodilo, in prevzamejo vso odgovornost.

2.2 Optimizacija skladiščnih sistemov

Kljub avtomatizacij skladiščnih sistemov je te še vedno možno dodatno optimizirati tako časovno kot energijsko, s tem pa se lahko dodatno zmanjšajo stroški skladiščnih sistemov ter poveča prepustnost skladišč. Učinkovitost skladiščnih sistemov je odvisna od [6] strukture skladiščnega sistema [7] in učinkovitosti skladiščnega procesa.

2.2.1 Struktura skladiščnega sistema

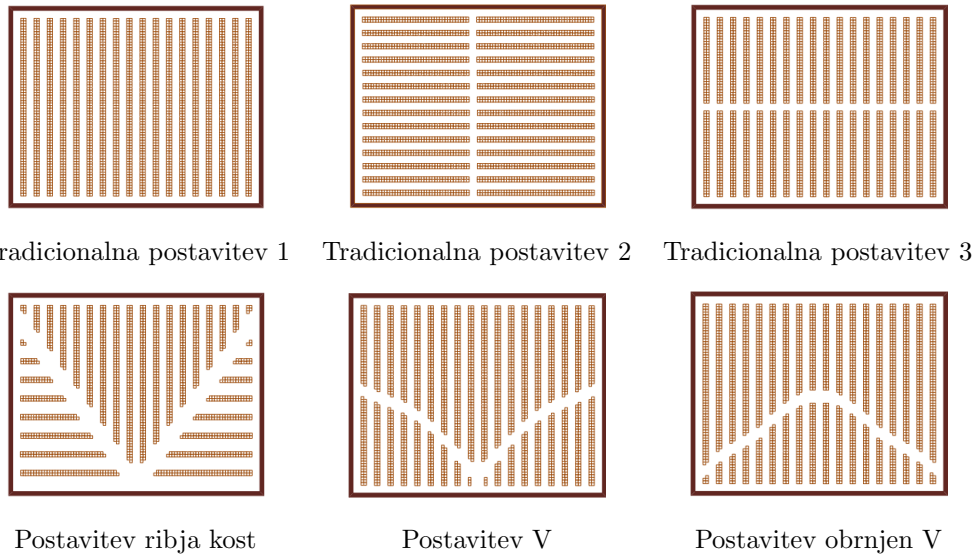
Struktura skladiščnega sistema zajema fizično postavitev celotnega skladišča ter specifikacije velikosti regalov in naprav v samem skladiščnem sistemu. Za uspešne nadaljnje optimizacije je zelo pomembno, da je skladišče postavljeno primerno našim zahtevam. Skladišče v osnovi sestavljajo:

- *Sprejemni prostor*: v njem se hranijo produkti, ko pridejo iz tovornega sredstva z namenom shranitve v skladišču. Produkti se tukaj zadržujejo, dokler se ustrezno ne obdelajo.
- *Skladiščni prostor*: tu so produkti dejansko shranjeni.
- *Izhodni prostor (ang.* staging*)*: v katerega gredo produkti, takoj ko so pobrani iz skladiščnega prostora. V njem so, dokler jih ne pripravijo za pošiljanje in odpošljejo stranki.
- *Pisarna*: služi nadzoru celotnega skladišča ter urejanju zadev v zvezi z naročili, s produkti, z dostavo ...

Vse te sklope je potrebno ustrezno povezati ter jih izdelati dovolj velike, da bodo ustrezali našim zahtevam. Pri skladiščnem prostoru je zelo pomembna tudi postavitev regalov, v katerih se bodo shranjevali produkti. Z dobro postavitvijo lahko že v tem koraku zmanjšamo čas pospravljanja in čas pobiranja produktov, ki ga bodo naprave potrebovale.

Postavitev regalov

Pri postavitvi regalov moramo upoštevati zahteve in pogoje, ki jih imamo zanje. Največkrat so regali postavljeni v kvadratni obliki in so tako posamezni tipi postavitev že standardizirani ter znane njihove prednosti in slabosti. Nekatere standardne postavitve regalov so prikazane na spodnji sliki 2.2.



Slika 2.2: Primer standardnih postavitv regalov [8]

Z različnimi postavitvami regalov lahko že v tem koraku zmanjšamo razdaljo, ki jo bodo morale naprave narediti, da bodo prišle do produktov. Pri opravilih, kjer morajo naprave iz neke začetne točke hoditi po produkte ter se nato vračati nazaj, postavitev V in postavitev ribja kost zmanjšata razdaljo, ki jo mora naprava opraviti, od 10 % pa do 20 %. Postavitev obrnjen V je od njiju boljša še za približno 3 %. Prava prednost takšnih postavitev pa se pozna pri nadaljnji optimizaciji v primerih, ko naprave pobirajo več produktov iz skladišča in jim je potrebno najti optimalno pot skozi skladiščni prostor. V takih primerih pride tudi do 60 % razlike v razdalji, ki jo morajo naprave opraviti med posameznimi postavitvami regalov v skladišču [8]. Žal pa zaradi regalnih konstrukcij avtomatskih regalnih sistemov pridejo v poštev le tradicionalne postavitve regalov, zato smo pri postavitvi regalov omejeni.

2.2.2 Optimizacija skladiščnega procesa

Pri avtomatskih skladiščnih sistemih s produkti rokujejo predvsem roboti, dvigala in druge avtomatske naprave, ki jih krmili sistem za upravljanje skladišča WMS (ang. *Warehouse Management System*). Takšni sistemi lahko delujejo po FIFO ali kakšni drugi enostavni metodi, lahko pa pri tem uporabljajo napredne metode, s katerimi optimizirajo kritične dele procesa. Kritična dela procesa glede na sistem WMS sta predvsem skladiščenje in pobiranje. Oba problema sta NP-težka, kar predstavlja pri optimizaciji dodaten izziv.

Skladiščenje

Glavni problem pri skladiščenju je dodeliti ustrezno lokacijo posameznim produktom. Cilj dobre dodelitve lokacij je, da bi naprave pri shranjevanju in pri pobiranju potrebovale čim manj časa, torej bi prepotovale čim krajšo razdaljo. Pri tem pa je želja, da bi bili produkti, ki bodo v istem naročilu, skupaj in bi se tako pobiranje izvršilo hitreje. Ker v večini primerov vnaprejšnja naročila niso znana, je pri tej optimizacijo potrebno izdelati nekakšen napovedni model in na podlagi njega določati mesta produktom.

Pobiranje

Pri pobiranju nastaneta dva NP-težka problema. Prvi je izbira skladiščnih mest, s katerih se bodo pobrali produkti v primeru več skladiščnih mest z istimi produkti. Drugi pa je vodenje naprave, da pridobi vse produkte z vseh izbranih skladiščnih mest. Pri izbiri skladiščnih mest je naš cilj izbrati takšna mesta, ki bi čim bolj olajšali vodenje naprav skozi skladišče. Za vodenje naprave skozi skladišče, da pobere vse produkte, se pri optimizaciji poti običajno uporabljajo algoritmi najkrajše poti v grafu [9].

2.3 Avtomatski skladiščni sistemi

Avtomatiziran shranjevanjevalni pridobitveni sistem (ang.*Automated storage and retrieval system*) je poseben tip skladišča, v katerem celotno delo s produkti opravljajo naprave, ki so vodene s strani sistema WMS. Takšno skladišče ima običajno V/I transportne poti ter skladiščna mesta, ki so namenjeni dejanskemu shranjevanju produktov. Tako je glavna naloga takšnih naprav le premikanje produktov iz V/I transportnih poti v skladiščna mesta ter obratno. Sistem WMS poskrbi za to, kam bo šel kateri produkt in kdaj. Tako sistem WMS napravam generira posebna opravila glede na naročila strank ter prejete pošiljke. Vsako generirano opravilo hrani začetno lokacijo produkta v skladišču ter končno lokacijo, kamor naj bi ga naprava prenesla, naloga naprav pa je, da to izvedejo. Pri tem se pojavita dva nova problema: vrstni red izvajanja opravil ter razporeditev opravil.

2.3.1 Vrstni red izvajanja opravil

V primeru, da ima sistem WMS več pripravljenih opravil za naprave v skladišču, lahko s pravilno razporeditvijo opravil običajno dodatno optimiziramo sistem. Cilj je, da opravila razvrstimo tako, da se naprave čim manj časa premikajo po skladiščnem prostoru brez produkta.

2.3.2 Razporeditev opravil

V primeru, da imamo v skladiščnem prostoru več naprav, ki premikajo produkte, je zelo pomembno, kako bomo med te naprave razdelili opravila, ki so ustvarjena s strani sistema WMS. Cilj je, da so napravam opravila razdeljena čim bolj enakomerno, da se med sabo pri opravljanju opravil ne ovirajo. Pomembno pa je tudi, da dobi posamezna naprava taka opravila, ki so relativno skupaj, saj potem naprave ni potrebno premikati skozi celotni skladiščni prostor. Ob izpolnitvi vseh treh zahtev dobimo največjo prepustnost skladiščnega sistema z nizkimi stroški.

2.4 Pregled sorodnih del

V članku [10] so se lotili optimizacije dodelitve lokacij produktom s hevristično metodo v kombinaciji z genetskim algoritmom. Skladišče, nad katerim so delali optimizacijo, je vsebovalo več skladiščnih naprav. Pri kriterijski funkciji genetskega algoritma so izhajali iz pričakovane obremenitve posamezne linije. Numerični testi so pokazali, da je z uporabo predlagane metode prepustnost skladišča večja kakor z uporabo naključne metode. Prav tako so hevristični algoritem za optimizacijo dodelitve lokacij produktov uporabili avtorji članka [11]. Vendar so v svoji raziskavi uporabili več različnih hevrističnih funkcij: evaluacija vseh možnih premikov praznih lokacij, maksimalni indeks (posebna funkcija pozicije skladiščnega mesta), najbližje prosto mesto, najbližje mesto končni lokaciji in genetski algoritem.

Optimizacijo razporejanja opravil dveh skladiščnih naprav z eno transportno skladiščno enoto so obravnavali v članku [12]. Problema so se lotili z dvema različnima matematičnima modeloma, opredeljenima z odločitvenimi spremenljivkami, ves čas pa so morali paziti tudi na to, da ni prišlo do trka med napravama. S takšnim pristopom so v določenih primerih dosegli tudi do 40 % pohitritev skladišča.

Optimizacija vrstnega reda pobiranja je opisana v članku [13]. Pri svoji optimizaciji so uporabili genetski algoritem. V funkciji uspešnosti so uporabili medsebojne razdalje v posebnem algoritmu, kar je doprineslo k bistvenemu izboljšanju v primerjavi z neoptimizirano potjo naprav.

Avtoji članka [14] so se lotili optimizacije poti skladiščne naprave. Njihov cilj je bil zmanjšati razdaljo, ki jo morajo naprave opraviti, da zberejo vse potrebne produkte za naročilo. Pri optimizaciji so uporabili metahevristične metode za reševanje problema trgovskega potnika (TSP). V povprečju jim je uspelo razdaljo, ki jo naprave opravijo, zmanjašti za 47 % z uporabo hevristične funkcije Lin–Kernighan–Helsgaun.

Poglavje 3

Uporabljene metode

Za optimizacijo problema razporejanja opravil dvigala smo uporabili dva različna pristopa, in sicer preiskovalni ter odločitveni pristop.

Pri preiskovalnem pristopu smo ustvarili graf vseh možnih poti, ki jih lahko dvigalo opravi, z upoštevanjem vseh omejitev. Nato pa smo z algoritmi najkrajše poti v grafu poiskali optimalno pot. Zaradi velike razsežnosti grafa ter zelo stroge omejitve časa preiskovanja smo uporabili več različnih algoritmov za iskanje najkrajše poti v grafu: Dijkstrov algoritem, Bellman-Fordov algoritem, genetski algoritem, optimizacijo s kolonijami mravelj, A*.

V primeru odločitvenega pristopa smo se v specifičnem stanju skladišča in dvigala za naslednje akcije odločali na podlagi opravil, ki so nam bile najbližje glede na trenutno lokacijo, dvigala. Poleg teh opravil smo upoštevali še opravila, ki so bila blizu poti, ki jo moramo opraviti v primeru, da imamo na dvigalu že kakšen produkt. V ta namen smo implementirali podatkovno strukturo KD-drevo, ki nam omogoča hitre dostope do bližnjih elementov v prostoru.

3.1 Dijkstrov algoritem

Dijkstrov algoritem išče najkrajšo pot v grafu. Uporablja se lahko za iskanje najkrajših poti od specifičnega vozlišča do vseh ostalih vozlišč[1], lahko pa se tudi predčasno prekine in se tako uporablja za iskanje najkrajše poti med dvema specifičnima vozliščima.

Dijkstrov algoritem sodi med najbolj znane in najpogosteje uporabljene algoritme za iskanje najkrajše poti med dvema vozliščima. Njegova edina omejitev je, da preiskovani graf ne sme vsebovati negativnih povezav. Časovna zahtevnost algoritma je $O(|V^2|)$, pri čemer je $|V|$ število vozlišč v grafu. Algoritem se lahko pohitri z uporabo prioritetne vrste za iskanje vozlišča z minimalno razdaljo od izvora. Tako se lahko algoritem pohitri na $O(|E| + |V| \log |V|)$, pri čemer je $|E|$ število povezav v grafu. To pa je asimptotično najhitrejše v primerjavi z ostalimi znanimi algoritmi za iskanje najkrajše poti med dvema vozliščema v usmerjenem grafu z nenegativnimi povezavami. V posebnih primerih grafa je mogoče Dijkstrov algoritem prilagoditi in s tem še dodatno izboljšati asimptotično zgornjo mejo. Primer dveh možnih izboljšav:

- uteži povezav so celoštevilske in navzgor omejene s konstanto $C \rightarrow$ implementacija z uporabo Fibonaccijeve kopice v kombinaciji s korensko kopico (ang. *radix heap*) $O(|E| + |V|\sqrt{\log |C|})$ [15],
- pri neusmerjenem grafu in celoštevilskih utežeh povezav je možno najti najkrajšo pot v linearnem času $O(|E| + |V|)$ [16].

[1] Zgradi drevo najkrajših poti

3.1.1 Algoritem

Ideja Dijkstrovega algoritma je relativno preprosta. Vozliščem dodelimo začetne razdalje, ki jih skozi postopek izboljšujemo, dokler s preiskovanjem ne obiščemo vseh vozlišč. Nato rekonstruiramo celotno drevo najkrajših poti od začetnega vozlišča do vseh ostalih na podlagi prej nastavljenih predhodnikov posameznih vozliščih. Postopek preiskovanja je naslednji:

1. Vsa vozlišča iz grafa damo v množico neobiskanih (Q), jim nastavimo razdaljo od začetnega vozlišča na neskončno (∞) ter njihove predhodnike na nedefinirane.

2. Začetnemu vozlišču nastavimo razdaljo na 0.

3. Iz množice neobiskanih vzamemo vozlišče (u), ki ima najmanjšo razdaljo. V prvem koraku bo to začetno vozlišče, ki smo mu v koraku 2 nastavili razdaljo na 0.

4. Vsem sosednjim elementom vozlišča u (S_u) popravimo razdaljo do začetnega vozlišča v primeru, da je ta preko vozlišča u krajša od dosedanje. Ravno tako popravimo predhodnika vozliščem S_u na vozlišče u.

5. Ponavljamo korak 3, dokler je v množici še kakšen element oziroma, v primeru da iščemo najkrajšo pot med dvema vozliščema, dokler iz množice ne dobimo ciljnega vozlišča.

6. Rekonstruiramo drevo najkrajših poti oziroma, v primeru iskanja poti med specifičnima vozliščema, določimo najkrajšo pot.

V psevdokodi algoritma (koda 1) je verzija algoritma, kjer iščemo najkrajšo pot med dvema določenima vozliščema. Predčasna zaključitev algoritma je razvidna v vrsticah 12 in 13, kjer se preverja, ali je trenutno obravnavano vozlišče že končno vozlišče. Prioritetno vrsto za dodatno pohitritev algoritma pa lahko uvedemo v vrstici 10, kjer iz množice vzamemo vozlišče z minimalno razdaljo.

Algoritem na koncu vrne razdalje vozlišč do začetnega vozlišča *distance*[] ter njihove predhodnike *predecessor*[], preko katerih je prišel s preiskovanjem. Na podlagi teh lahko rekonstruiramo celotno najkrajšo pot med izbranima vozliščema (*source*, *target*).

Koda 1 Psevdokoda Dijkstrovega algoritma

```
1:  function DIJKSTRA(graph, source, target)
2:      create vertex set Q
3:      create dictionaries distance[], predecessor[]
4:      for all vertex v in Graph do
5:          distance[v] ← INFINITY
6:          predecessor[v] ← UNDEFINED
7:          add v to Q
8:
9:      distance[source] ← 0
10:     while Q is not empty do
11:         u ← vertex in Q with min distance[u]
12:         if u == target then
13:             break
14:         remove u from Q
15:
16:         for all neighbor v of u do
17:             alt ← distance[u] + length(u, v)
18:             if alt < distance[v] then
19:                 distance[v] ← alt
20:                 predecessor[v] ← u
        return distance[], predecessor[]
21: end function
```

3.2 Bellman-Fordov algoritem

Tudi Bellman-Fordov algoritem je algoritem za iskanje najkrajše poti v grafu, vendar ga lahko za razliko od Dijkstrovega algoritma uporabljamo tudi na grafih z negativnimi povezavami. Algoritem je bil opisan s strani več avtorjev, prvi ga je predlagal Shimbel leta 1955, nato ga je objavil Bellman leta 1958. Kasneje, leta 1959, pa je isti algoritem objavil še Moore [17].

Algoritem uporablja pristop dinamičnega programiranja, pri čemer iterativno izboljšuje rešitve celotnega grafa na podlagi predhodnih rešitev. Ker v vsaki iteraciji optimizira nad celotnim grafom, je časovno tudi zahtevnejši v primerjavi z Dijkstrovim algoritmom. Njegova asimptotična zgornja meja je $O(|E||V|)$. Poleg možnosti iskanja najkrajše poti v grafu z negativnimi povezavami lahko algoritem uporabimo tudi za iskanje negativnih ciklov[2] [18], saj se njegovo izvajanje v primeru, da ga najde, zaključi. Asimptotične zgornje meje pri tem algoritmu ni možno izboljšati, tako kot pri Dijkstrovem algoritmu, so pa možne modifikacije, ki v praktičnih primerih pohitrijo delovanje algoritma kljub enaki zgornji asimptotični meji. Yen je predstavil dve modifikaciji, s pomočjo katerih je možno zmanjšati število iteracij glavne zanke, v najslabšem primeru z $|V|-1$ na $\frac{|V|}{2}$ [19]. Poleg te modifikacije lahko uvedemo še naključne porazdelitve, s čimer postane pričakovano število iteraciji glavne zanke $\frac{|V|}{3}$, kar v praksi predstavlja že veliko pohitritev.

3.2.1 Algoritem

Bellman-Fordov algoritem deluje ravno tako kakor Dijkstrov algoritem na principu iterativnega izboljševanja rešitve. Algoritem se v vsaki iteraciji $|V|-1$ krat sprehodi skozi vse povezave v grafu. Pri obravnavanju posamezne povezave v iteraciji preveri ter po potrebi izboljša razdaljo med vozlišči obravnavane povezave. V vsaki iteraciji se število vozlišč s pravilno razdaljo

[2]Negativen cikel na grafu je cikel, ki ima vsoto razdalj povezav negativno (torej manjšo od 0).

povečuje, moramo pa algoritem vedno izvesti do konca, ker ne vemo, kdaj je kakšno vozlišče že zaključeno.

Potek algoritma:

1. Vsem vozliščem iz grafa nastavimo razdaljo od začetnega vozlišča na neskončno(∞) ter njihove predhodnike na nedefinirane.
2. Začetnemu vozlišču nastavimo razdaljo na 0.
3. $|V|-1$-krat pregledamo vse povezave ter popravimo razdalje in predhodnike do ponora povezav v primeru, ko je preko trenutno obravnavane povezave pot bolj ugodna od dosedanje.
4. Še enkrat se sprehodimo skozi vse povezave in preverimo, ali so možne izboljšave, krajše poti do kateregakoli vozlišča. V primeru možnih izboljšav v tem koraku graf vsebuje vsaj en negativni cikel, kar sporočimo kot napako.
5. V primeru, da graf ne vsebuje negativnih ciklov, rekonstruiramo drevo najkrajših poti na podlagi nastavljenih predhodnikov.

Psevdokoda algoritma (koda 2) kot vhodna parametra prejme graf (*graph*), v katerem bo algoritem iskal najkrajšo pot, in izvorno vozlišče (*source*), od katerega bo iskal razdalje do vseh ostalih. Za razliko od psevdokode Dijkstrovega algoritma ta metoda nima ponora, ker algoritem ne more iskati najkrajše poti med dvema izbranima vozliščema, ampak jo mora poiskati za vsa vozlišča v grafu. Dokler ne opravi vseh iteracij, nima nobenih podatkov, do katerega vozlišča že ima najdeno najkrajšo pot.

Preden algoritem zaključi, preveri, ali obstajajo negativni cikli. Na koncu algoritem vrne razdalje vozlišč od začetnega vozlišča *distance*[] ter njihove predhodnike *predecessor*[]. Na podlagi teh lahko rekonstruiramo celotno drevo najkrajših poti med začetnim vozliščem ter vsemi ostalimi.

Koda 2 Psevdokoda Bellman-Fordovega algoritma

```
1:  function BellmanFord(graph, source)
2:      create dictionaries distance[], predecessor[]
3:      verttices ← list of all vertices in graph
4:      edges ← list of all edges in graph
5:
6:      for all vertex v in verttices do
7:          distance[v] ← INFINITY
8:          predecessor[v] ← UNDEFINED
9:
10:     dist[source] ← 0
11:     for i from 1 to size(verttices)-1 do
12:         for all edge(u,v) with weight w in edges do
13:             if distance[u] + w < distance[v] then
14:                 distance[v] ← distance[u] + w
15:                 predecessor[v] ← u
16:
17:     for all edge(u,v) with weight w in edges do
18:         if distance[u] + w < distance[v] then
19:             return error: "Graph contains a negative-weight cycle"
        return distance[], predecessor[]
20: end function
```

3.3 Genetski algoritem

Genetski algoritem je hevristično preiskovalni algoritem, ki nam ne zagotavlja optimalne rešitve, nam pa zato lahko vrne približno rešitev v krajšem času, kakor jo vrnejo algoritmi, ki nam zagotavljajo optimalne rešitve. Kako dobra je rešitev, ki jo algoritem najde, lahko posredno nadzorujemo preko parametrov algoritma in s tem neposredno vplivamo na njegov čas izvajanja. Algoritem je splošno zastavljen tako, da se ga lahko uporablja za različne preiskovalne namene, med drugimi tudi za iskanje najkrajše poti v grafu. Ideja evolucijskega reševanja inženirskih problemov se je pojavila sredi dvajsetega stoletja. Takrat se je pričelo več neodvisnih računalniških znanstvenikov ukvarjati z implementacijo naravne evolucije v računalništvu. Njihov cilj je bil izboljšati rešitev (nekega problema) na podoben način, kakor to stori narava s selekcijo osebkov v populaciji [20].

3.3.1 Osnovni koncept algoritma

Genetski algoritem simulira naravno evolucijo tako, da najprej inicializira začetno populacijo kromosomov, za katere izračuna uspešnost (ang. *fitness*) s pomočjo kriterijske funkcije. Vsak kromosom predstavlja veljavno rešitev problema. Nato se v vsaki iteraciji izvedejo genetske operacije na trenutni populaciji kromosomov [21]. Genetske operacije so:

- križanje, s katerim na podlagi izbranih dveh kromosomov ustvarimo nova kromosoma, ki sta sestavljena iz elementov osnovnih kromosomov,
- mutacija, s katero iz enega kromosoma naredimo drugega,
- selekcija, pri kateri izberemo kromosome, ki bodo tvorili novo populacijo v naslednji iteraciji.

Naštete genetske operacije ponavljamo, dokler nismo zadovoljni z rešitvijo oziroma dokler ne presežemo časovne omejitve. Na koncu vrnemo kromosom, ki je najboljši v populaciji. Psevdokoda algoritma je prikazana v kodi 3.

Koda 3 Psevdokoda genetskega algoritma

```
1: function GENETIC
2:     t ← 0
3:     initPopulation[P(t)]
4:     evalPopulation[P(t)]
5:     repeat
6:         M ← applyMutation(P(t))
7:         C ← applyCrossower(P(t))
8:         evalPopulation(C, M)
9:         P(t + 1) ← selection(P(t), C, M))
10:        t ← t + 1
11:    until break OR the result is good enough return best(P(t))
12: end function
```

Kodiranje kromosomov

Kromosomi so pogosto predstavljeni kot niz bitov, nad katerim je možno enostavno izvajati križanje in mutacije [22]. Na sliki 3.1 je prikazan bitno kodiran kromosom, kodiranje je lahko tudi drugačno, vendar moramo potem ustrezno prilagoditi funkcijo križanja in funkcijo mutacije.

Slika 3.1: Primer kromosoma

Kromosom 1	0 1 0 0 1 0 0 0 1 0 1 1 1

Inicializacija začetne populacije

Velikost začetne populacije je odvisna od problema, ki ga optimiziramo, oziroma od dolžine kromosomov. Zelo priporočljivo je, da je populacija čim večja, da se začetni kromosomi dobro razpršijo po celotnem preiskovalnem prostoru, vendar to lahko zelo upočasni delovanje algoritma. Zato moramo pri izbiri velikosti začetne populacije najti kompromis med kakovostjo rešitev in časom izvajanja algoritma. Začetno populacijo lahko ustvarimo naključno ali pa s hevrističnimi metodami [23]. Pri naključni inicializaciji se začetni kromosomi ustvarijo naključno – naključne rešitve našega problema. Pri hevristični inicializaciji se za začetne kromosome izbere kromosome z visoko uspešnostjo – samo dobre rešitve problema. Z izbranimi kromosomi glede na uspešnost algoritem usmerimo v preiskovalnem prostoru, kar posledično pripomore k hitrejši konvergenci kromosomov. Vendar pri takšni začetni populaciji lahko zelo hitro obstanemo v lokalnem minimumu in ne najdemo globalnega. Z naključno inicializacijo je verjetnost, da obstanemo v lokalnem minimumu, manjša.

Križanje

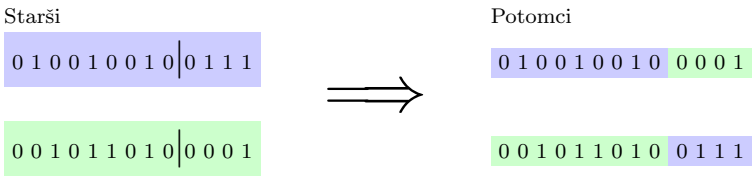
Metodo križanja uporabljamo, da izmenjujemo informacijo med dvema kromosomoma (staršema) in s tem pridobivamo nove kromosome (potomce), ki so morda boljši od izvornih [24]. Starša, nad katerima se izvede križanje, naključno izberemo iz populacije. Število križanj je zopet odvisno od problema; za izbrani problem moramo določiti smiselno vrednost.

Obstaja zelo veliko operacij križanja kromosomov, med njimi so najbolj znane:

- *Enotočkovno križanje*
  Pri enotočkovnem križanju se izbere ena naključna točka, na kateri se razdelita oba starša. Potomci so nato ustvarjeni iz kombinacije staršev, na primer potomec je sestavljen iz prvega dela prvega starša ter drugega

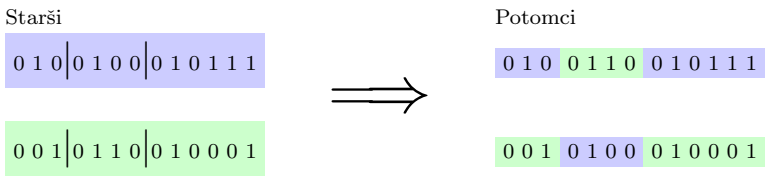
dela drugega starša, kakor je prikazano na sliki 3.2.

Slika 3.2: Enotočkovno križanje



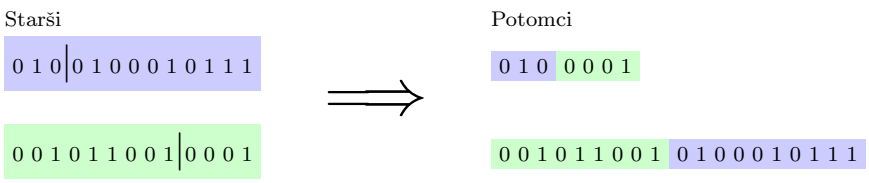
- *Dvotočkovno križanje*
  Pri dvotočkovnem križanju se izbereta dve naključni točki. Pri kreiranju potomcev si vmesni del med tema točkama starša izmenjata, kakor je prikazano na sliki 3.3.

Slika 3.3: Dvotočkovno križanje



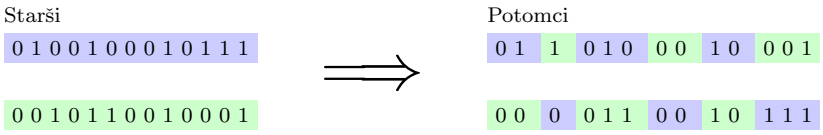
- *Prepleteno križanje (ang. cut and splice)*
  Pri prepletenem križanju se vsakemu staršu naključno določi točka, na kateri se bo delil. Običajno sta ti dve točki različni in s tem nastanejo potomci, ki so kombinacija staršev, različnih dolžin, kot je prikazano na sliki 3.4.

Slika 3.4: Prepleteno križanje



- *Enakomerno križanje*
  Pri enakomernem ali uniformnem križanju so biti naključno skopirani iz prvega ali iz drugega starša v potomce. Pri tem je možno definirati razmerje mešanja staršev, s čimer imamo večji nadzor nad križanjem. Primer 50 % križanja je prikazan na sliki 3.5, kjer ima vsak potomec enako število bitov od obeh staršev.

Slika 3.5: Enakomerno križanje



Mutacija

Mutacija je metoda, ki z določeno verjetnostjo kromosome naključno spremeni. Z mutacijo poskrbimo za ohranitev raznolikosti med kromosomi ter s tem preprečujemo, da bi se optimizacija ustavila v lokalnem minimumu [23]. Del kromosoma, ki mutira, je naključno izbran, verjetnost mutacije pa je običajno zelo nizka, $0,1\%$. Primeri nekaj možnih mutacij kromosomov:

- *Inverzija bitov* (ang. *flip bit*)
  Zamenja izbrani bit $0 \rightarrow 1$ in obratno $1 \rightarrow 0$.

- *Omejitev* (ang. *boundary*)
  V primeru številskega kodiranja kromosomov se lahko uporabi mutacija omejitev. Mutacija zamenja vrednost s spodnjo ali z zgornjo mejo, kar je možno v danem kontekstu.

- *Neenakomerno* (ang. *non-uniform*)
  Verjetnost mutiranja kromosomov se skozi generacije spreminja.

- *Gaussova* (ang. *Gaussian*)
  Naključnim vrednostim kromosoma se doda Gaussovo porazdeljeno naključno vrednost.

Uspešnost, kriterijska funkcija

Kriterijska funkcija je specifična za vsak problem posebej, njena naloga pa je izračun uspešnosti posameznih kromosomov. Funkcija mora biti enostavna oziroma hitro izračunljiva. V nasprotnem primeru, ko je kriterijska funkcija časovno zahtevna, se lahko uporablja tudi aproksimacija uspešnosti namesto natančne vrednosti. Na podlagi izračunanih vrednosti se v nadaljevanju kromosome primerja med sabo.

Selekcija

Selekcija je postopek, s katerim izberemo novo populacijo za naslednjo iteracijo. Nova populacija se izbira izmed vseh kromosomov trenutne populacije, mutiranih kromosomov ter potomcev, ki smo jih pridobili s križanjem kromosomov v trenutni iteraciji [25]. Metod za izbiranja kromosomov je zelo veliko, nekatere izmed njih so:

- *Selekcija z ruleto*
  Pri selekciji z ruleto ima vsak posamezni kromosom dodeljeno verjetnost izbora, ki je proporcionalna njegovi uspešnosti. Na podlagi teh verjetnosti je potem izbrana nova populacija.

- Turnirska selekcija
  Kromosome se naključno izbere v paru, nato pa se v novo populacijo doda le boljšega izmed njiju.

- *Delež najboljših*
  Novo populacijo sestavljajo le kromosomi, ki so najbolj uspešni.

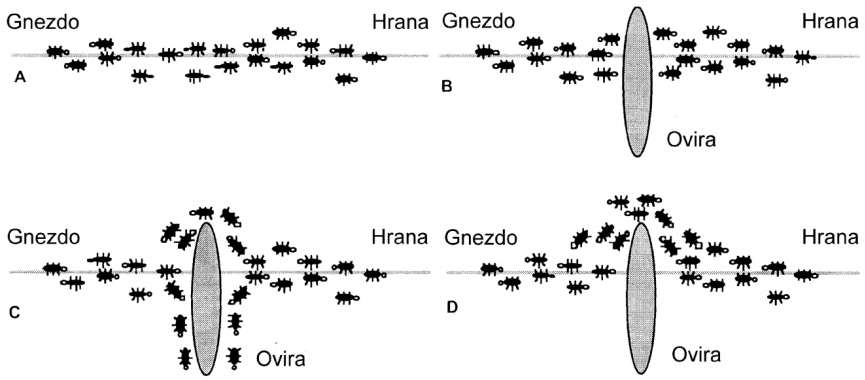
- *Naključna*
  Kromosomi za novo populacijo so naključno izbrani med vsemi kromosomi.

3.4 Optimizacija s kolonijami mravelj

Optimizacija s kolonijami mravelj je metahevristična metoda za reševanje optimizacijskih problemov. Enako kakor genetski algoritmi nam tudi ta metoda ne zagotovlja optimalne rešitve problema. Metoda posnema kolonije mravelj v naravi, ki so sposobne brez vizualnih informacij najti najkrajšo pot od izvora do cilja.

Delo na metodi s posnemanjem kolonij mravelj sta pričela S. Goss in Jean-Louis Deneuborg leta 1989. V svojih raziskavah sta ugotovila, da so mravlje sposobne najti najkrajšo pot med gnezdom in virom hrane brez uporabe vizualnih receptorjev; pri iskanju poti si pomagajo s spuščanjem kemičnih snovi – feromonov [26]. Sam algoritem s kolonijami mravelj je prvi zasnoval Marco Dorigo za reševanje problema najkrajše poti v grafu [27].

Na sliki 3.6 je prikazan osnovni koncept premikanja mravelj, po katerem je izdelan algoritem. Na začetku mravlje hodijo po hrano in jo prinašajo nazaj v gnezdo po ravni črti, kakor je prikazano na sliki 3.6a. Mravlje po svoji poti spuščajo feromone, ki s časom izhlapevajo. Te feromone mravlje uporabljajo tudi za koordinacijo oziroma navigacijo, kam morajo iti, saj jim sledijo in tako ostanejo na pravi poti. V primeru postavitve ovire 3.6b na njihovo pot se sled feromonov, ki jo imajo mravlje, prekine. Za obnovitev sledi feromonov in posledično poti morajo mravlje najti novo pot od gnezda do hrane. V iskanju nove poti se mravlje enakomerno porazdelijo po obeh možnih poteh 3.6c in sproti po poti spuščajo nove feromone. Mravlje, ki so izbrale krajšo pot, bodo hitreje prišle na sled starim feromonom in tako združile pot nazaj v celoto. Mravlje, ki gredo v nasprotno stran, bodo zaznale večjo količino feromonov na krajši poti, zato se bodo odločile za stran, ki je krajša in pot še bolj ojačale. Posledično bodo vse mravlje pričele hoditi po krajši poti od gnezda do hrane 3.6d.



Slika 3.6: Potovanje mravelj [26][28]

3.4.1 Algoritem

Jedro algoritma je povzeto po obnašanju mravelj v naravi, ki si informacijo o poteh izmenjujejo preko feromonov. Na začetku je potrebno algoritmu inicializirati iskalne mravlje, ki bodo iskale rešitve v grafu. V vsaki iteraciji glavne zanke programa se mravlje premikajo po grafu in iščejo nove rešitve. Na koncu iteracije pa si mravlje izmenjajo informacije o poteh preko feromonov, pri čemer je potrebna osvežitev vseh feromonov na vseh poteh [29]. Osnovni koraki algoritma so:

- inicializacija mravelj,
- premik mravelj in
- posodobitev feromonov.

Pri vsakem izmed teh korakov moramo ustrezno nastaviti posamezne atribute za optimalno reševanje danega problema. Vrednosti so specifične za vsak problem posebej in z njimi tudi vplivamo na kakovost rešitev ter časovno izvajanje algoritma. Za bolj temeljito preiskovanje algoritmu lahko dodamo še lokalno preiskovanje. Psevdokoda algoritma je prikazana v kodi 4.

Koda 4 Psevdokoda algoritma kolonija mravelj

```
1: function AntColony
2:     ants ← initializeAnts()
3:     path_gb ← null
4:     repeat
5:         paths_i ← {}
6:         for all ant in ants do
7:             path_l ← moveAnts(ant)
8:             if path_l != null then
9:                 paths_i ← paths_i ∪ path_l
10:                if path_l better than globalBestPath then
11:                    path_gb ← path_l
12:        pheromoneUpdate(paths_i)
13:    until break OR the result is good enough return path_gb
14: end function
```

Inicializacija

Na začetku moramo inicializirati vse mravlje na začetno točko v grafu. Število mravelj je poljubno in izbrati moramo smiselno vrednost. V primeru premajhnega števila mravelj se te ne bodo mogle dobro razpršiti po celotnem grafu in algoritem ne bo prišel do izraza. V primeru prevelikega števila mravelj pa se bo le povečal čas izvajanja algoritma. Poleg inicializacije mravelj moramo na začetku inicializirati še feromone na vseh povezavah v grafu na začetne vrednosti τ_0. Začetna vrednost feromonov je parameter algoritma, ki ga ustrezno prilagodimo našemu problemu [30].

Izbira naslednje povezave

Mravlje v vsakem koraku izbirajo naslednje povezave na podlagi feromonov, ki so že na teh povezavah, ter cene, ki jo ima posamezna povezava. Verjetnost, da bo določena mravlja izbrala specifično povezavo xy, je podana v enačbi 3.1 [31], pri čemer so njeni parametri:

- τ_{xy}, trenutna količina feromonov na povezavi xy,
- η_{xy}, priljubljenost povezave xy, običajno obratna vrednost cene povezave c_{xy}, $\eta_{xy} = \frac{1}{c_{xy}}$,
- α, parameter, s katerim nadzorujemo vpliv feromonov na izbiro povezave, običajno je v območju $0 \leq \alpha$,
- β, parameter, s katerim kontroliramo vpliv cene povezave na izbiro povezave, običajno je v območju $1 \leq \beta$,
- successors(x), množica vseh naslednjih vozlišč, do katerih lahko pridemo iz vozlišča x preko direktne povezave.

$$p_{xy} = \frac{\tau_{xy}^{\alpha} \eta_{xy}^{\beta}}{\sum_{z \in successors(x)} (\tau_{xz}^{\alpha} \eta_{xz}^{\beta})} \quad . \tag{3.1}$$

Posodobitev feromonov

Po končanem premikanju mravelj je potrebno pred novo iteracijo premikanja mravelj posodobiti feromone na vseh povezavah [32]. Feromoni na povezavah se posodobijo glede na enačbi 3.2 in 3.3, pri čemer so njuni parametri:

- p, faktor izhlapevanja feromonov skozi iteracije,
- m, število vseh mravelj, ki preiskujejo graf,
- $\Delta\tau_k(x, y)$, dodatna vrednost feromonov, ko je mravlja k odkrila pot od izvora do cilja in šla skozi povezavo xy v trenutni iteraciji,
- L_k, cena celotne poti, ki jo je našla mravlja k.

$$\tau_{xy} = (1 - p)\tau_{xy} + \sum_{k=1}^{m} \Delta\tau_k(x, y) \quad , \tag{3.2}$$

$$\Delta\tau_k(x, y) = \begin{cases} \frac{1}{L_k}, & \text{če je šla mravlja skozi povezavo } xy \\ 0, & \text{sicer} \end{cases} \quad . \tag{3.3}$$

3.5 Algoritem A*

Algoritem A* sodi med najbolj znane preiskovalne algoritme v umetni inteligenci [33]. Kakor genetski algoritem 3.3 in optimizacija s kolonijami mravelj 3.4 je tudi algoritem A* hevristični preiskovalni algoritem. Za razliko od njiju lahko pri algoritmu A* preko hevristične funkcije določimo, ali nam bo vrnil optimalno rešitev problema ali hevristični približek. Z delom na algoritmu A* je pričel Nils Nilsson, ko je želel leta 1968 optimizirati pot robota, ki se premika po sobi z ovirami [34].

3.5.1 Koncept algoritma

Sam algoritem je v osnovi le razširjena verzija Dijkstrovega algoritma 3.1, tako da časovna zahtevnost algoritma v najslabšem primeru ostaja enaka kot pri Dijkstrovem algoritmu, ki je $O(|E| + |V| \log |V|)$ ob uporabi prioritetne vrste. Algoritem A* deluje po metodi najprej najboljši (ang. *best-first*), ki pri raziskovanju grafa razišče najprej najbolj obetavna vozlišča [35]. Priljubljenost (ceno) določenega vozlišča (u) računa po enačbi

$$f(u) = g(u) + h(u) \quad , \tag{3.4}$$

pri čemer je $g(u)$ natančna razdalja od izvora do vozlišča u, funkcija $h(u)$ pa je hevristična ocena razdalje od vozlišča u do cilja in predstavlja jedro samega algoritma.

Hevristična funkcija algoritma

S hevristično funkcijo določamo hitrost in velikost območja, ki ga bo algoritem preiskal, od nje pa je odvisna tudi časovna zahtevnost algoritma. Tako lahko z njo določamo natančnost oziroma hitrost samega algoritma. Posebni primeri hevristične funkcije so:

- $h(u) = 0$
  Hevristična ocena razdalje do cilja je za vsa vozlišča 0. V tem primeru bo algoritem izbiral naslednja vozlišča le na osnovi funkcije $g(u)$, kar pretvori algoritem nazaj v Dijkstrov algoritem, s čimer imamo zagotovljeno optimalno rešitev.

- $h(u) \leq h^*(u)$
  Hevristična ocena razdalje je vedno manjša ali enaka dejanski razdalji ($h^*(u)$) od vozlišča do cilja.

- $\exists u : h(u) > h^*(u)$
  Hevristična ocena razdalje je včasih večja od dejanske razdalje. V takšnih primerih hevristične funkcije nam algoritem ne zagotavlja več optimalne rešitve, vendar se bo skozi graf premikal hitreje, torej bo čas izvajanja krajši.

Za zagotavljanje popolnosti algoritma[3] mora biti hevristična funkcija za vsa vozlišča dosledna (ang. *admissible*), monotona ter nenegativna [36]. To pomeni, da mora za vsa vozlišča v grafu veljati:

- *Doslednost*: $h(x) \leq d(x, y) + h(y)$, pri čemer ima vozlišče x povezavo do vozlišča y.

- *Monotonost*: $h(v1) \leq d(v1, v2) + h(v2) \leq d(v1, v2) + d(v2, v3) + h(v3)$, pri čemer ima vozlišče $v1$ povezavo do vozlišča $v2$, vozlišče $v2$ pa povezavo do vozlišča $v3$.

- *Nenegativnost*: $h(x) \geq 0$.

[3] Algoritem je popoln, kadar vrne optimalno rešitev

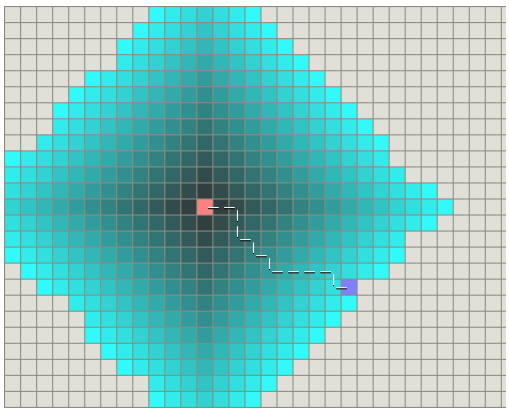
Primerjava z Dijkstrovim algoritmom

Na naslednjih slikah (3.7, 3.8, 3.9, 3.10) je prikazana razlika v velikosti preiskovalnega prostora pri Dijkstrovem algoritmu ter algoritmu A* [37]. Pri opravljeni primerjavi je algoritem A* za hevristično funkcijo uporabljal funkcijo Manhattanske razdalje, ki ustreza vsem pogojem za popolnost algoritma A*. Manhattanska razdalja med dvema točkama (P_1, P_2) v dvodimenzionalnem prostoru se izračuna po enačbi

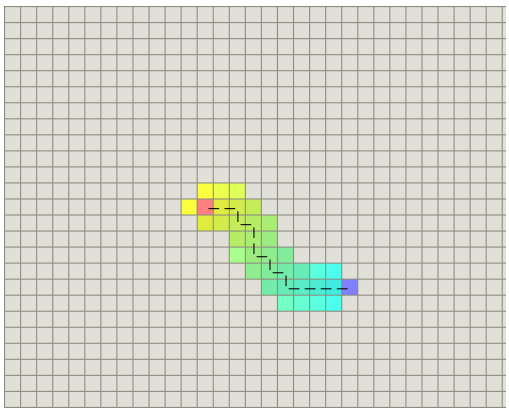
$$f(P_1, P_2) = |X(P_1) - X(P_2)| + |Y(P_1) - Y(P_2)| \quad . \tag{3.5}$$

pri čemer sta $X(P), Y(P)$ funkciji, ki vrneta posamezni koordinati točk. Cilj obeh algoritmov je bil najti optimalno pot od roza kvadratka do modrega kvadratka na prikazanih slikah. Pri Dijkstrovem algoritmu je z odtenkom barve nakazan čas razvoja vozlišč, pri algoritmu A* pa vrednost hevristične ocene za dano polje.

V prvi primerjavi je izvedeno preiskovanje najkrajše poti v prostoru brez ovir. Dijkstrov algoritem (slika 3.7) je moral za doseganje optimalne poti preiskati vsa vozlišča v radiju iskanega vozlišča. Algoritem A* pa je bil zaradi hevristične funkcije dobro usmerjen in je posledično razvil le vozlišča, ki so bila v poti ciljnega vozlišča (slika 3.8).

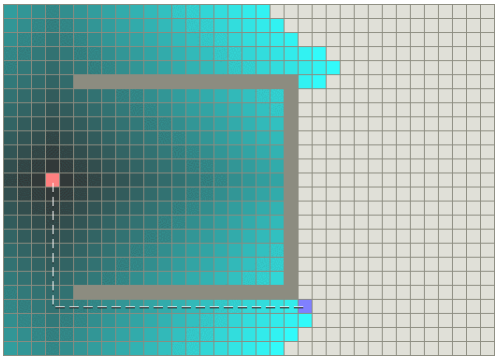


Slika 3.7: Dijkstrov algoritem

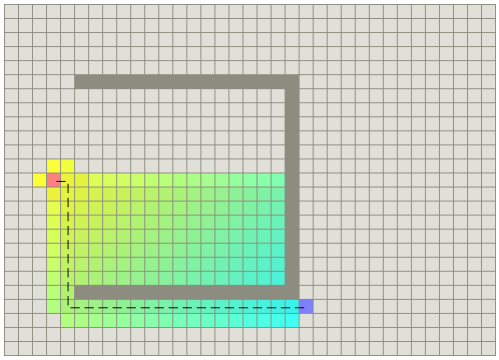


Slika 3.8: Algoritem A*

Pri drugi primerjavi pa je prikazano preiskovanje algoritmov v prostoru z oviro. Dijkstrov algoritem še vedno preišče skoraj enako veliko območje (slika 3.9), kakor ga je pred postavitvijo ovire. V primeru, da bi bila ovira v celoti znotraj radija iskanega vozlišča, bi bilo to območje popolnoma enako veliko z oviro ali brez nje. Algoritmu A* pa se preiskovano območje zelo poveča (slika 3.10), vendar je še vedno za pol manjše od območja, ki ga je preiskal Dijkstrov algoritem.



Slika 3.9: Dijkstrov algoritem



Slika 3.10: Algoritem A*

3.5.2 Algoritem

Algoritem A* pri svojem preiskovanju uporablja odprto in zaprto množico vozlišč. V množicah hrani vozlišča, ki jih je že preiskal, ter tista, ki jih še mora [38]. Na začetku v množico odprtih vstavi le začetno vozlišče, kasneje pa se skozi postopek sproti dodajajo še ostala vozlišča v grafu. Jedro algoritma A* deluje tako, da se iz odprte množice vozlišč jemlje vozlišča po prioriteti, ki je izračunana po hevristični funkciji algoritma. Vozlišča jemlje iz odprte množice, vse dokler ne pride do ciljnega vozlišča oziroma se odprta množica ne izprazni. Pri obravnavi posameznega vozlišča posodobi ustrezne razdalje sosednjih vozlišč, ki še niso v zaprti množici, tako, da na novo preračuna hevristično oceno in jih vstavi v odprto množico, če le ta še niso bila v odprti množici. Trenutno obravnavano vozlišče pa doda v zaprto množico, tako da se nanj ne bo več vračal. Psevdokoda algoritma je prikazana v kodi 5.

Koda 5 Psevdokoda algoritma A*

```
1: function A*(graph, source, target)
2:     create vertex set closedSet, openSet
3:     create dictionaries disFromStart[],disToTarget[], predecessor[]
4:     for all vertex v in Graph do
5:         disFromStart[v] ← disToTarget[v] ← INFINITY
6:         predecessor[v] ← UNDEFINED
7:         add v to openSet
8:     disFromStart[source] ← 0
9:     disToTarget[source] ← heuristicDis(source, target)
10:    while openSet is not empty do
11:        u ← vertex in openSet with min disToTarget[u]
12:        if u == target then
13:            break
14:        remove u from openSet
15:        insert u into closedSet
16:        for all neighbor v of u do
17:            if vinclosedSet then
18:                continue
19:            disFromStartThruUToV ← disFromStart[u] + length(u, v)
20:            if vnotinopenSet then
21:                add v to openSet
22:            else if disFromStartThruUToV >= disFromStart[v] then
23:                continue
24:            predecessor[v] ← u
25:            disFromStart[v] ← disFromStartThruUToV
26:            disToTarget[v] ← disFromStart[v] + heuristicDis(v, target)
27:    return disFromStart[], predecessor[]
28: end function
```

3.6 KD-drevo

KD-drevo je binarna drevesna podatkovna struktura, ki omogoča hitro iskanje najbližjih sosednjih elementov v poljubno dimenzionalnem prostoru. Sodi med najbolj uporabljane podatkovne strukture pri iskanju najbližjih sosednjih elementov (ang. *nearest neighbor search*) [39]. Podatkovno strukturo je izumil Jon Louis Bentley leta 1975 [40]. Časovne zahtevnosti njenih osnovnih operacij so:

- izgradnja drevesne strukture:
  - $O(n \log(n))$ v primeru uporabe algoritma za iskanje mediane v vsaki iteraciji izgradnje,
  - $O(n \log^2 n)$ v primeru iskanje mediane s predhodnim sortiranjem elementov z algoritmom časovne zahtevnosti $O(n \log(n))$ (na primer urejanje z združevanjem (ang. *mergesort*), urejanje s kopic (ang. *heapsort*)).
- iskanje enega sosednjega elementa v poravnanem drevesu: $O(\log(n))$,
- vstavljanje novega elementa v poravnano drevo: $O(\log(n))$,
- odstranitev naključnega elementa iz poravnanega drevesa: $O(\log(n))$.

Pri tem n pa predstavlja velikost problema, število elementov v podatkovni strukturi. Takšne časovne zahtevnosti so zelo dobre za operacije, ki jih podatkovna struktura izvaja. V primeru neporavnane drevesne strukture so lahko v najslabšem primeru časovne zahtevnosti za operacije iskanja, vstavljanja in odstranjevanja ranga $O(n)$, vendar to lahko preprečimo s sprotnim prilagajanjem drevesne strukture ob vstavljanju ter odstranjevanju elementov.

3.6.1 Osnovne operacije

Podatkovna struktura KD-drevo je binarno drevo, ki hrani v vsakem vozlišču naslednje parametre [41]:

- *dim*: predstavlja dimenzijo, po kateri so se elementi razdelili na trenutnem nivoju,
- *point*: element, s katerim se primerjajo vsi ostali elementi na trenutnem nivoju po dimenziji *dim*,
- *left*: levo podvozlišče, ki vsebuje vse elemente, ki so manjši od trenutnega vozlišča v dimenziji, ki je na trenutnem nivoju,
- *right*: desno podvozlišče, ki vsebuje vse elemente, ki so večji od trenutnega vozlišča v dimenziji, ki je na trenutnem nivoju.

Elementi so v drevesni strukturi urejeni glede na koordinate v k-dimenzionalnem prostoru. Ker je KD-drevo binarna podatkovna struktura, so lahko na enem nivoju drevesa elementi urejeni le po eni dimenziji. Zaradi tega KD-drevo za vsak naslednji nivo podvozlišč izbere drugo/naslednjo dimenzijo, po kateri bodo urejena njegova podvozlišča. Tako vsako vozlišče predstavlja središčno točko za vsa njegova podvozlišča glede na izbrano dimenzijo. Vsi elementi, ki so glede na izbrano dimenzijo večji, se bodo nahajali v desni veji, vsi manjši pa v levi veji drevesa.

Izgradnja drevesne strukture

Pri izgradnji podatkovne strukture na podlagi seznama vseh elementov, ki jih želimo shraniti, na vsakem nivoju vse elemente razdelimo glede na mediano v trenutni dimenziji. Tako imamo zagotovljeno uravnoteženo drevo, ki je optimalno za našo podatkovno strukturo.

Dodajanje elementov

Pri dodajanju novega elementa v podatkovno strukturo poiščemo njegovo mesto. Iskanje mesta pričnemo pri korenskem vozlišču in se pomikamo proti

listom drevesa na podlagi dimenzij posameznih vozlišč. Ko pridemo do lista drevesa, vanj ustrezno dodamo nov element. Pri tem se nam lahko drevo izrodi in je potrebno opraviti dodatno poravnavo drevesa.

Odstranjevanje elementov

Pri brisanju elementa najprej poiščemo ustrezni element, ga odstranimo ter v primeru, da ni bil listni element, nadomestimo z njegovim ustreznim podvozliščem. Pri tem poskrbimo, da se ne poruši urejenost vozlišč. Tudi v tem primeru se nam lahko drevo izrodi, zato je potrebna dodatna poravnava drevesne strukture za ohranitev poravnanosti drevesa.

Poravnava drevesa

Za poravnavo drevesa uporabimo osnovne tehnike za rotacijo drevesnih struktur. Pri tem je potrebno paziti, da ohranimo pravilno razporeditev vseh vozlišč glede na izbrane dimenzije. Takšna operacija je lahko pri podatkovni strukturi KD-drevo zelo zahtevna, saj je včasih potrebno preurediti večji del poddrevesa, da ohranimo poravnanost.

Iskanje sosednjih elementov v izbranem območju

Pri iskanju sosednjih elementov v izbranem območju pričnemo zopet pri korenskem vozlišču in se pomikamo proti listom. Za trenutno obravnavano vozlišče preverimo, ali je znotraj iskanega območja ali ne. V primeru, da je, vstavimo njegov element v seznam najbližjih elementov, ki ga na koncu vrnemo kot rezultat metode. Nato postopek rekurzivno ponavljamo za njegova podvozlišča, dokler ne pridemo do listov. Pri tem še dodatno preverimo, ali res obstaja možnost, da se sosednji elementi v izbranem območju nahajajo v obeh podvozliščih, glede na trenutno oddaljenost od elementa. Kadar zagotovo vemo, da v določenem podvozlišču ne bomo našli sosednjih elementov v izbranem območju, v to podvozlišče ne gremo več iskat in si s tem zmanjšamo preiskovalni prostor in čas iskanja.

3.6.2 Algoritem

Metoda za izgradnjo podatkovne strukture KD-drevo (koda 6) kot vhodni parameter prejme seznam elementov, ki jih želimo vanjo shraniti, ter dimenzijo trenutnega nivoja (*depth*). Dimenzija trenutnega nivoja se z vsakim nivojem poveča za ena po modulu k, ki predstavlja dimenzijo prostora, v katerem so elementi razpršeni. Začetna dimenzija ni tako pomembna, običajno je to dimenzija 0. Kot rezultat algoritem vrne korensko vozlišče drevesa, ki vsebuje vse elemente, ki smo jih želeli vanj shraniti.

Koda 6 Psevdokoda izgradnja podatkovne strukture KD-drevo

```
1:  function KDTree(pointList, dept)
2:      axis ← depth mod k
3:      medianPoint ← selectMedianByAxis(pointList, axis)
4:      pointsBefore ← getPointsBefore(pointList, medianPoint)
5:      pointsAfter ← getPointsAfter(pointList, medianPoint
6:      node ← newInstance()
7:      node.dim ← axis
8:      node.point ← medianPoint
9:      if length(pointsBefore) > 0 then
10:         node.left ← KDTree(pointsBefore, depth + 1)
11:     if length(pointsAfter) > 0 then
12:         node.righ ← KDTree(pointsAfter, depth + 1)
13:     return node
14: end function
```

Poglavje 4

Obravnavano skladišče

Skladiščni sistem, ki smo ga optimizirali, je avtomatiziran shranjevalno-pridobitveni sistem. Za delo s produkti skrbi ena sama naprava - dvigalo, ki se pri svojih operacijah giblje le v dveh dimenzijah. Posebnost dvigala v izbranem skladiščnem sistemu v primerjavi z ostalimi skladiščnimi sistemi je ta, da ima to dvigalo na voljo dve togo sklopljeni mesti za transportne skladiščne enote.

V skladiščnem sistemu so na posameznih nivojih različne proizvodne linije, ki dobivajo surovine in polizdelke iz skladišča in izdelke v skladišče vračajo. Skladišče je tako namenjeno shranjevanju surovin, končnih izdelkov in polizdelkov (izdelkov ene linije, ki gredo na vhod naslednje linije). Zato tudi taka nekonvencionalna razporeditev V/I mest, prikazana na slikah 4.1 in 4.2. Ker bo zaradi vseh opravil dvigalo v končni fazi precej obremenjeno, je pomembno, da je dobro optimizirano.

4.1 Regalna konstrukcija

Izbrani skladiščni sistem vsebuje dva regala s 36 skladiščnimi mesti po dolžini ter 102 po višini. Nekatera skladiščna mesta niso na voljo za shranjevanje zaradi V/I mest, ki so postavljena med samimi skladiščnimi mesti. Vsa V/I mesta imajo dodatno mesto v globini, kar je predstavljalo dodatno omejitev pri optimizaciji opravil.

4.1.1 Oznake mest

Za označevanje posameznih mest v skladiščnem sistemu se uporabljajo tri vrste oznak.

Oznaka za skladiščno mesto je **RrXxxYyyy**, pri čemer

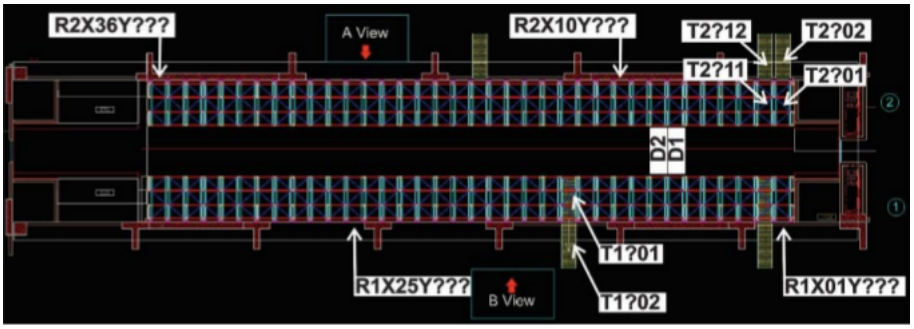
- **r** predstavlja indeks regala 1...2,
- **xx** predstavlja horizontalni indeks 01...36,
- **yyy** predstavlja vertikalni indeks 001...102.

Oznaka za V/I mesto je **Trlxz**, pri čemer

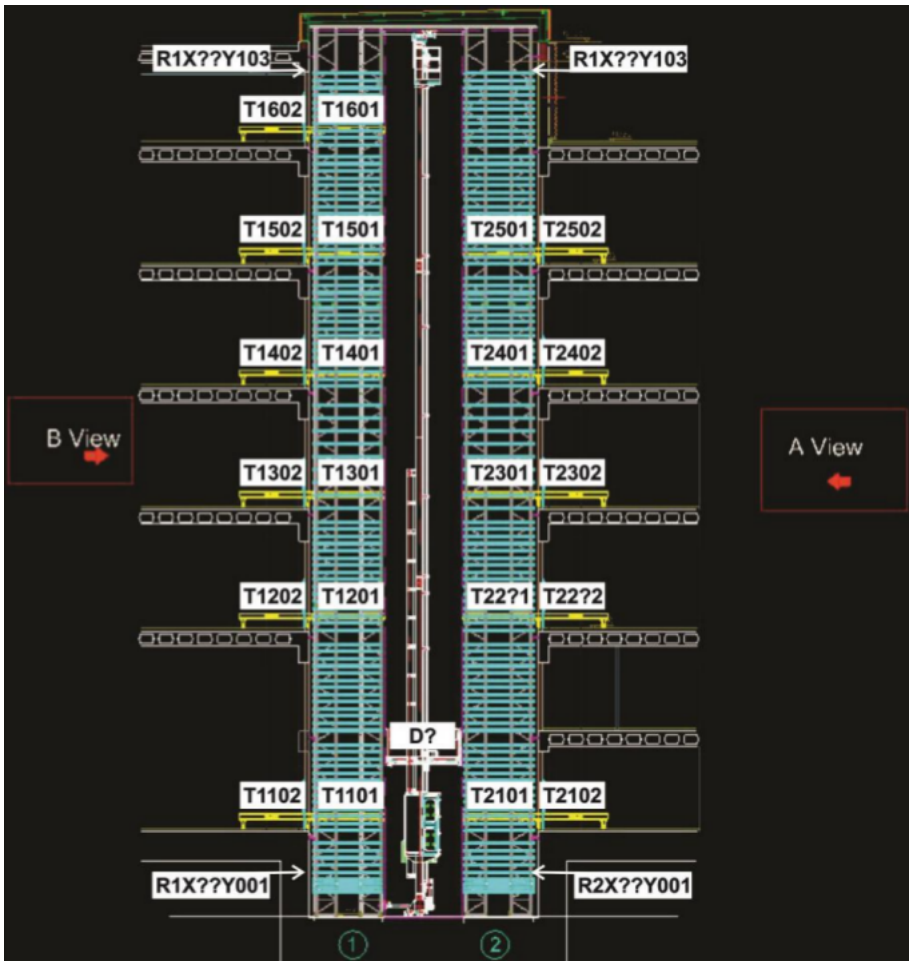
- **r** predstavlja indeks regala 1...2,
- **l** predstavlja nivo 1...6,
- **x** predstavlja horizontalni indeks 0...1,
- **z** predstavlja globino 1...2.

Oznaka posameznega mesta za transportno skladiščno enoto na dvigalu (mizo) je **Dd**, pri čemer

- **d** predstavlja indeks posamezne mize na dvigalu 1...2.



Slika 4.1: Shema skladiščnega sistem od zgoraj



Slika 4.2: Shema skladiščnega sistema s prednje strani

4.2 Dvigalo

Dvigalo ima na voljo dve mesti za transportne skladiščne enote. Na teh dveh mestih lahko skladiščne operacije izvaja hkrati, kar pripomore k dodatni optimizaciji celotnega sistema. Parametri dvigala, ki smo jih upoštevali pri nadaljnjih optimizacijskih kalkulacijah, so prikazani v spodnji tabeli 4.1.

Tabela 4.1: Tabela podanih parametrov sistema

oznaka	opis	vrednost	enota
v_{max_x}	Maksimalna hitrost v horizontalni smeri	1,5	$[\frac{m}{s}]$
a_{max_x}	Maksimalni pospešek v horizontalni smeri	0,5	$[\frac{m}{s^2}]$
j_x	Trzaj v horizontalni smeri	1	$[\frac{m}{s^3}]$
m_x	Masa v horizontalni smeri	15000	$[kg]$
J_x	Vztrajnostni moment v horizontalni smeri	0,1586	$[kgm^2]$
d_x	Premer kolesa v horizontalni smeri	0,4	$[m]$
i_x	Prestavno razmerje v horizontalni smeri	13,74	$[-]$
η_x	Izkoristek v horizontalni smeri	0,86	$[-]$
k_x	Koeficient trenja v horizontalni smeri	0,09	$[-]$
v_{max_y}	Maksimalna hitrost v vertikalni smeri	0,85	$[\frac{m}{s}]$
a_{max_y}	Maksimalni pospešek v vertikalni smeri	0,5	$[\frac{m}{s^2}]$
j_y	Trzaj v vertikalni smeri	0,5	$[\frac{m}{s^3}]$
m_y	Masa v vertikalni smeri	1550	$[kg]$
J_y	Vztrajnostni moment v vertikalni smeri	0,1473	$[kgm^2]$
d_y	Premer bobna za navijanje jeklenice v vertikalni smeri	0,705	$[m]$
i_y	Prestavno razmerje v vertikalni smeri	62,2	$[-]$
η_y	Izkoristek v vertikalni smeri	0,84	$[-]$
k_y	Koeficient trenja v vertikalni smeri	0,09	$[-]$
t_{oi}	Čas premika transportno skladiščne enote z zunanjega V/I mesta v notranje ali obratno	8	$[s]$
t_{pr}	Čas izvajanja operacije nalaganje/razlaganje	8,23	$[s]$
t_{change}	Minimalen čas vstavljanja/odstranjevanja skladiščno transportne enote v/s sistema	60	$[s]$
$t_{process}$	Minimalni čas obdelave skladiščno transportne enote	30	$[s]$
η_{zw}	Učinkovitost izmenjave energije	0,98	$[-]$
P_{0inv}	Poraba frekvenčnih pretvornikov	500	$[W]$
P_{0aux}	Poraba ostalih naprav	500	$[W]$
g	Gravitacijska konstanta	0,98	$[\frac{m}{s^2}]$

4.2.1 Čas gibanja

Za potrebe čim bolj realne optimizacije smo izračunali natančen čas, ki ga dvigalo potrebuje pri premiku z enega mesta v skladišču na drugega. Pri gibanju dvigala v skladiščnem sistemu smo upoštevali tudi trzaj. Tako zato smo za izračun natančnega časa uporabili osnovne enačbe za pospešek $a(t)$ in hitrost $v(t)$ pri neenakomernem pospešenem gibanju.

$$j(t) = \frac{da(t)}{dt} \quad , \tag{4.1}$$

$$a(t) = \frac{dv(t)}{dt} \quad , \tag{4.2}$$

$$v(t) = \frac{dx(t)}{dt} \quad . \tag{4.3}$$

V našem primeru smo imeli konstanten trzaj, tako da smo vse ostale enačbe izpeljali iz njega. Do pospeška smo prišli z integracijo trzaja, do hitrosti z integracijo pospeška in do razdalje $x(t)$ z integracijo hitrosti, pri čemer smo upoštevali morebitne začetne vrednosti pospeška a_0, hitrosti v_0 in razdalje x_0.

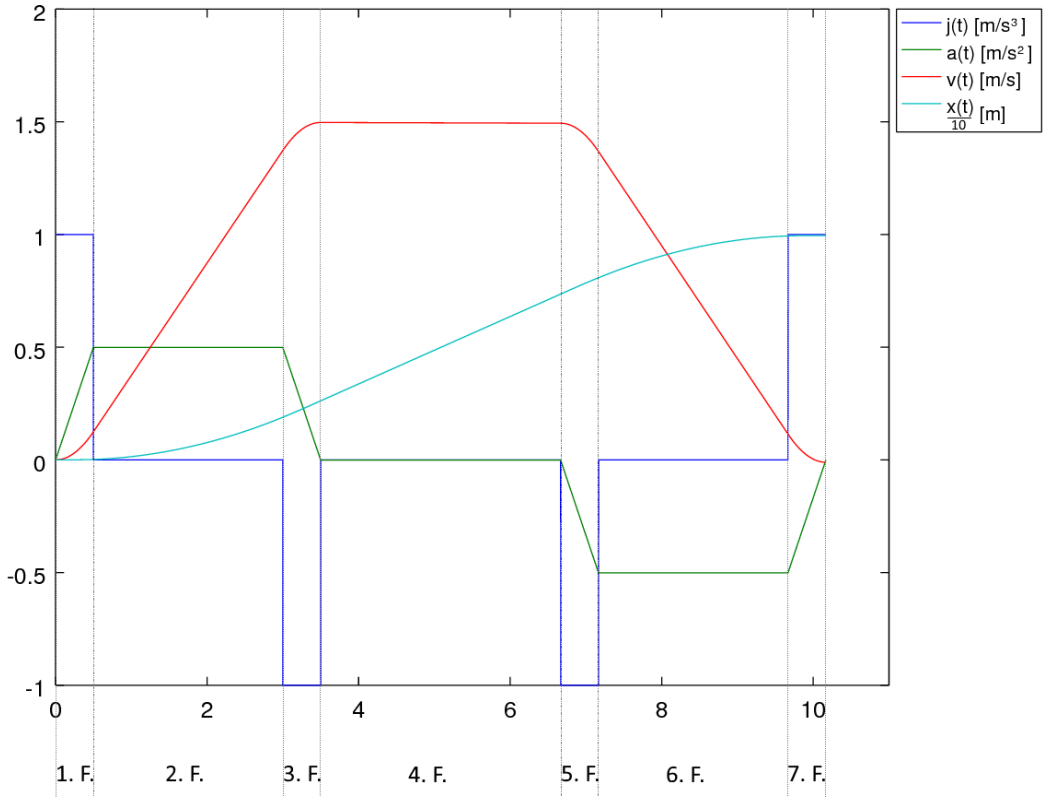
$$a(t) = a_0 + \int_0^t j dt = a_0 + jt \quad , \tag{4.4}$$

$$v(t) = v_0 + \int_0^t a(t) dt = v_0 + a_0 t + \frac{jt^2}{2} \quad , \tag{4.5}$$

$$x(t) = x_0 + \int_0^t v(t) dt = x_0 + v_0 t + \frac{a_0 t^2}{2} + \frac{jt^3}{6} \quad . \tag{4.6}$$

Na sliki 4.3 so prikazane funkcije poti, hitrosti, pospeška ter trzaja pri premiku dvigal v horizontalni smeri za 10 m.
Pri gibanju dvigala tako nastopa 7 faz:

1. pospeševanje do maksimalnega pospeška ($j = \text{konst.}$),
2. pospeševanje s konstantnim pospeškom ($a = \text{konst.}$),
3. pojemanje do pospeška nič ($j = \text{konst.}$),
4. gibanje z maksimalno hitrostjo ($v = \text{konst.}$),
5. pojemanje do minimalnega pospeška ($j = \text{konst.}$),
6. pojemanje s konstantnim pospeškom ($a = \text{konst.}$),
7. pospeševanje do pospeška nič ($j = \text{konst.}$).



Slika 4.3: Graf poti, hitrosti, pospeška in trzaja v odvisnosti od časa

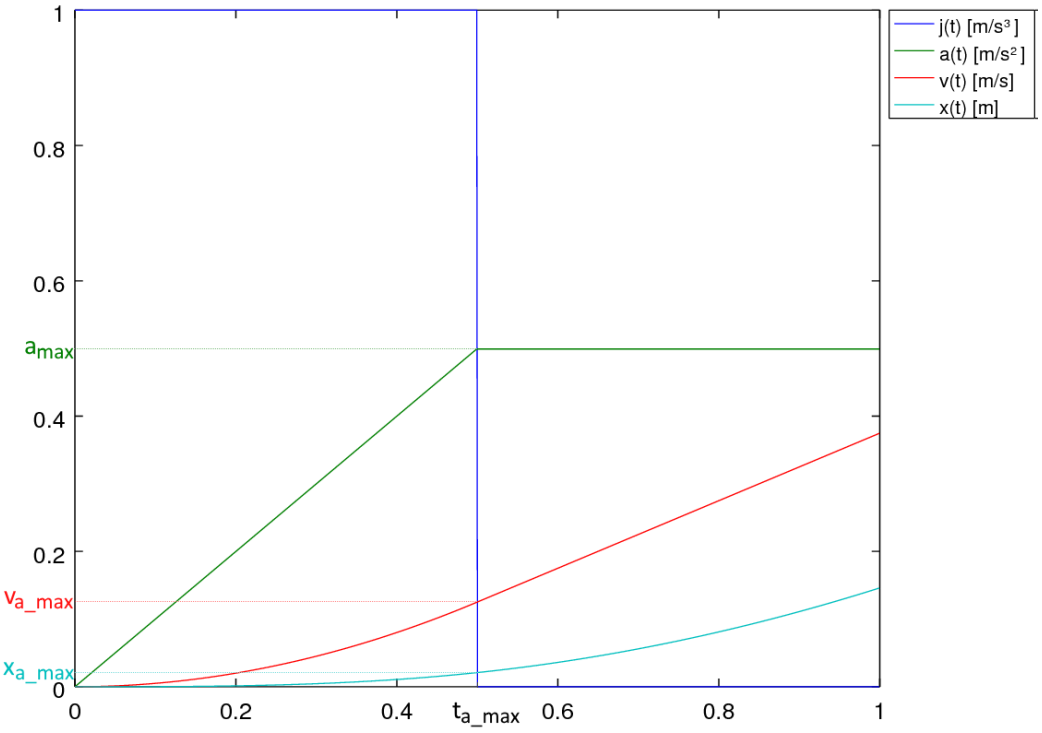
Skupni čas, ki ga dvigalo potrebuje za prvo, drugo in tretjo fazo, je enak času, ki ga dvigalo potrebuje za peto, šesto in sedmo fazo gibanja. Ravno tako je enaka skupna razdalja, ki jo opravi dvigalo v prvi, drugi in tretji fazi, skupni razdalji pete, šeste in sedme faze gibanja. Z upoštevanjem tega se izračuni časa za opravljanje določene razdalje poenostavijo. Pri računanju časa nastopajo tako trije robni primeri:

- Primer 1: Razdalja, ki jo mora dvigalo opraviti, je prekratka, da bi dvigalo razvilo svoj maksimalni pospešek.
- Primer 2: Razdalja, ki jo mora dvigalo opraviti, je prekratka, da bi dvigalo razvilo svojo maksimalno hitrost.
- Primer 3: Razdalja, ki jo mora dvigalo opraviti, je dovolj dolga za razvoj maksimalne hitrosti dvigala.

Za določitev primera, po katerem bomo računali čas, ki ga dvigalo potrebuje za svojo razdaljo, moramo najprej izračunati minimalno razdaljo za razvoj maksimalnega pospeška ter minimalno razdaljo za razvoj maksimalne hitrosti. Pri naslednjih enačbah bomo uporabili splošne oznake za trzaj j, pospešek a_{max} in hitrost v_{max}, ki jih pri računanju želenega gibanja (horizontalnega ali vertikalnega) ustrezno nadomestimo z j_x, a_{max_x} ,v_{max_x} pri horizontalnem gibanju oziroma j_y, a_{max_y} ,v_{max_y} pri vertikalnem gibanju.

Minimalna razdalja za maksimalni pospešek

Za izračun minimalne razdalje za doseganje maksimalnega pospeška x_{limit1}, moramo izračunati razdaljo, ki je potrebna za gibanje, sestavljeno le iz prve, tretje, pete in sedme faze gibanja. Pri tem sta prva in tretja faza po času in razdalji ekvivalentna fazama pet in sedem. Tako je dovolj, da izračunamo le razdaljo, ki je potrebna za prvo in tretjo fazo, ter dobljeni rezultat podvojimo za izračun celotnega gibanja.



Slika 4.4: Graf poti, hitrosti, pospeška in trzaja v odvisnosti od časa

Na sliki 4.4 je prikazana prva sekunda funkcij poti, hitrosti, pospeška ter trzaja pri premiku dvigal v horizontalni smeri za 10 m. Na njej so dodatno označene vrednosti:

- x_{a_max} - opravljena pot ob dosegu maksimalnega pospeška,
- v_{a_max} - dosežena hitrost ob dosegu maksimalnega pospeška,
- t_{a_max} - čas, pri katerem dosežemo maksimalni pospešek,
- a_{max} - maksimalni pospešek.

Prva faza: Pospeševanje do maksimalnega pospeška
Pri prvi fazi smo najprej po osnovnih enačbah za neenakomerno pospešeno gibanje izračunali čas, ki je potreben za doseganje maksimalnega pospeška t_{a_max}, nato pa še hitrost v_{a_max}, ki jo dosežemo ob dosegu maksimalnega pospeška, ter razdaljo x_{a_max}, ki jo pri tem opravimo.

$$t_{a\_max} = \frac{a_{max}}{j} \quad , \tag{4.7}$$

$$v_{a\_max} = v_0 + a_0 t + \frac{jt^2}{2} = 0 + 0 + \frac{j(\frac{a_{max}}{j})^2}{2} = \frac{a_{max}^2}{2j} \quad , \tag{4.8}$$

$$x_{a\_max} = x_0 + v_0 t + \frac{a_0 t_{a\_max}^2}{2} + \frac{j t_{a\_max}^3}{6} = 0 + 0 + 0 + \frac{j t_{a\_max}^3}{6} \quad , \tag{4.9}$$

$$x_{a\_max} = \frac{j(\frac{a_{max}}{j})^3}{6} = \frac{a_{max}^3}{6j^2} \quad . \tag{4.10}$$

Tretja faza: Pojemanje do pospeška nič

Čas pojemanja do pospeška nič t_{a_zero} pri tretji fazi je enak času pospeševanja v prvi fazi. Pri računanju celotne razdalje prve in tretje faze x_{13F} upoštevamo opravljeno razdaljo ter hitrost ob koncu prve faze.

$$t_{a\_zero} = t_{a\_max} = \frac{a_{max}}{j} \quad , \tag{4.11}$$

$$x_{13F} = x_0 + v_0 t + \frac{a_0 t_{a\_max}^2}{2} + \frac{j t_{a\_max}^3}{6} \quad , \tag{4.12}$$

$$x_{13F} = x_{a\_max} + v_{a\_max} t_{a\_max} + \frac{a_{max} t_{a\_max}^2}{2} + \frac{j t_{a\_max}^3}{6} \quad , \tag{4.13}$$

$$x_{13F} = \frac{1}{6}\frac{a_{max}^3}{j^2} + \frac{1}{2}\frac{a_{max}^2}{j}\frac{a_{max}}{j} + \frac{a_{max}(\frac{a_{max}}{j})^2}{2} - \frac{j(\frac{a_{max}}{j})^3}{6} \quad , \tag{4.14}$$

$$x_{13F} = \frac{a_{max}^3}{j^2} \quad . \tag{4.15}$$

Skupaj

Celotna minimalna razdalja za doseganje maksimalnega pospeška $x_{total\_for\_a\_max}$ je sestavljena iz vseh štirih zgoraj naštetih faz. Po koncu tretje faze imamo le skupno razdaljo prve in tretje faze gibanja, zato moramo za celotno gibanje dobljeno razdaljo pri tretji fazi podvojiti.

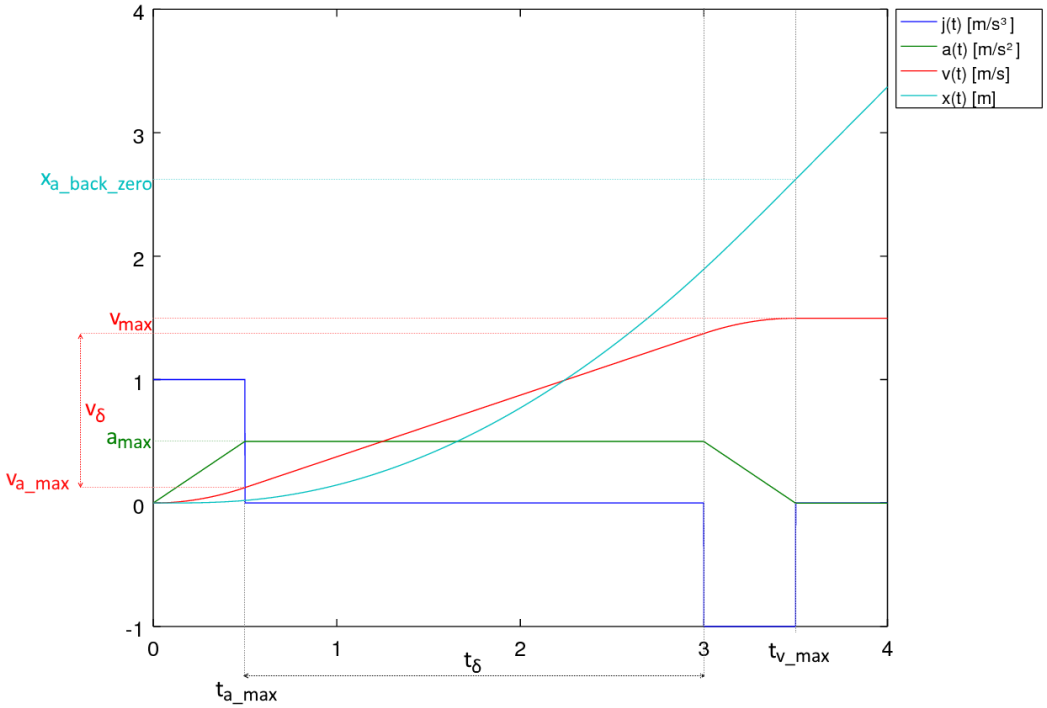
$$x_{limit1} = x_{total\_for\_a\_max} = 2\frac{a_{max}^3}{j^2} \quad . \tag{4.16}$$

Minimalna razdalja za maksimalno hitrost

Za izračun minimalne razdalje za doseganje maksimalne hitrosti moramo izračunati razdaljo, ki je potrebna za gibanje, sestavljeno le iz prve, druge, tretje, pete, šeste in sedme faze gibanja. Pri tem so prva, druga in tretja faza po času in razdalji ekvivalentne fazam pet, šest in sedem. Tako je dovolj, da izračunamo le razdaljo, ki je potrebna za prvo, drugo in tretjo fazo, ter dobljeni rezultat podvojimo za izračun celotnega gibanja.

Na sliki 4.5 so prikazane prve štiri sekunde funkcij poti, hitrosti, pospeška ter trzaja pri premiku dvigal v horizontalni smeri za 10 m. Na njej so dodatno označene vrednosti:

- a_{max} – maksimalni pospešek,
- v_{a_max} – dosežena hitrost ob dosegu maksimalnega pospeška,
- v_{max} – maksimalna hitrost,
- v_{δ} – razlika hitrosti pri konstantnem pospeševanju,
- t_{a_max} – čas, pri katerem dosežemo maksimalni pospešek,
- t_{v_max} – čas, pri katerem dosežemo maksimalno hitrost,
- t_{δ} – trajanje konstantnega pospeševanja,
- $x_{a\_back\_zero}$ – opravljena pot ob dosegu maksimalne hitrosti.



Slika 4.5: Graf poti, hitrosti, pospeška in trzaja v odvisnosti od časa

Prva faza: Pospeševanje do maksimalnega pospeška
Za izračun časa t_{a_max}, hitrosti v_{a_max} ter razdalje x_{a_max}, ko dosežemo maksimalni pospešek, uporabimo osnovne enačbe.

$$t_{a\_max} = t_{a\_zero} = \frac{a_{max}}{j} \quad , \tag{4.17}$$

$$v_{a\_max} = \frac{a_{max}^2}{2j} \quad , \tag{4.18}$$

$$x_{a\_max} = \frac{a_{max}^3}{6j^2} \quad . \tag{4.19}$$

Druga faza: Pospeševanje do skoraj maksimalne hitrosti
Pri drugi fazi najprej izračunamo razliko hitrosti v_δ, enakomernega pospeševanja, nato njej pripadajoč čas t_δ ter na koncu še skupno razdaljo x_{12F} prve in druge faze. Pri čemer potrebujemo za izračun razlike hitrosti izračunati, še razliko

hitrosti, ki jo opravimo pri pojemanju iz maksimalnega pospeška do pospeška nič v_{a_zero}.

$$v_{a\_zero} = \frac{a_{max}^2}{2j} = v_{a\_max} \quad , \tag{4.20}$$

$$v_\delta = v_{max} - v_{a\_max} - v_{a\_zero} = v_{max} - 2v_{a\_max} = v_{max} - \frac{a_{max}^2}{j} \quad , \tag{4.21}$$

$$t_\delta = \frac{v_\delta}{a_{max}} = \frac{1}{a_{max}} \left(v_{max} - \frac{a_{max}^2}{j} \right) = \frac{v_{max}}{a_{max}} - \frac{a_{max}}{j} \quad , \tag{4.22}$$

$$x_{12F} = x_{a\_max} + v_{a\_max} t_\delta + \frac{a_{max} t_\delta^2}{2} \quad , \tag{4.23}$$

$$x_{12F} = \frac{a_{max}^3}{6j^2} + \frac{a_{max}^2}{2j} \left(\frac{v_{max}}{a_{max}} - \frac{a_{max}}{j} \right) + \frac{a_{max} (\frac{v_{max}}{a_{max}} - \frac{a_{max}}{j})^2}{2} \quad , \tag{4.24}$$

$$x_{12F} = \frac{a_{max}^3}{6j^2} - \frac{a_{max} v_{max}}{2j} + \frac{v_{max}^2}{2a_{max}} \quad . \tag{4.25}$$

Tretja faza: Pojemanje do pospeška nič

Razdalja, ki jo opravimo v tretji fazi $x_{a\_back\_zero}$, je sestavljena iz razdalje prve in druge faze, pri računanju pa moramo upoštevati tudi končno hitrost $v_{max} - v_{a\_max}$, ki jo dosežemo v drugi fazi gibanja.

$$x_{a\_back\_zero} = x_{delta} + v_{stop\_a} t_{a\_zero} + \frac{a_{max} t_{a\_zero}^2}{2} - \frac{j t_{a\_zero}^3}{6} \quad , \tag{4.26}$$

$$\begin{aligned} x_{a\_back\_zero} = & \left(\frac{a_{max}^3}{6j^2} - \frac{a_{max} v_{max}}{2j} + \frac{v_{max}^2}{2a_{max}} \right) + \\ & \left(v_{max} - \frac{a_{max}^2}{2j} \right) \frac{a_{max}}{j} + \frac{a_{max} (\frac{a_{max}}{j})^2}{2} - \frac{j (\frac{a_{max}}{j})^3}{6} \quad , \end{aligned} \tag{4.27}$$

$$x_{a\_back\_zero} = \frac{v_{max}^2}{2a_{max}} + \frac{v_{max} a_{max}}{2j} \quad . \tag{4.28}$$

Skupaj

Tudi v tem primeru moramo za izračun celotne minimalne razdalje za doseganje maksimalne hitrosti $x_{total\_for\_v\_max}$ končno razdaljo tretje faze podvojiti. S tem upoštevamo tudi zadnje tri faze gibanja.

$$\begin{aligned} x_{limit2} = x_{total\_for\_v\_max} = 2x_{a\_back\_zero} &= 2\left(\frac{v_{max}^2}{2a_{max}} + \frac{v_{max}a_{max}}{2j}\right) \\ &= \frac{v_{max}^2}{a_{max}} + \frac{v_{max}a_{max}}{j} \quad . \end{aligned} \tag{4.29}$$

Primer 1

Pri primeru 1 računamo čas, ki je potreben, da dvigalo opravi razdaljo s_1, ki je prekratka, da bi dvigalo razvilo svoj maksimalni pospešek a_{max}. Za izračun časa moramo najprej izračunati, kolikšen pospešek lahko na izbrani razdalji dosežemo a_{r_max}. Po izračunanem pospešku izračunamo še čas iz pospeška $t_{a_{r\_max}}$, ki predstavlja $\frac{1}{4}$ celotnega časa gibanja. Gibanje je zopet sestavljeno iz prve, tretje, pete in sedme faze gibanja, pri čemer sta prva in tretja faza zopet ekvivalentni peti in sedmi fazi glede na čas in razdaljo.

Računanje maksimalnega pospeška

$$t_{a_{r\_max}} = \frac{a_{r\_max}}{j} \quad , \tag{4.30}$$

$$x_{1F} = x_0 + v_0 t + \frac{a_0 t^2}{2} + \frac{jt^3}{6} = 0 + 0 + 0 + \frac{j(\frac{a_{r\_max}}{j})^3}{6} = \frac{a_{r\_max}^3}{6j^2} \quad , \tag{4.31}$$

$$v_{1F} = v_0 + a_0 t + \frac{jt^2}{2} = 0 + 0 + \frac{j(\frac{a_{r\_max}}{j})^2}{2} = \frac{a_{r\_max}^2}{2j} \quad , \tag{4.32}$$

$$\frac{s_1}{2} = x_{1F} + v_{1F} t_{a_{r\_max}} + \frac{a_{r\_max} t_{a_{r\_max}}^2}{2} - \frac{j t_{a_{r\_max}}^3}{6} \quad , \tag{4.33}$$

$$\frac{s_1}{2} = \frac{a_{r\_max}^3}{6j^2} + \frac{a_{r\_max}^2}{2j}\frac{a_{r\_max}}{j} + \frac{a_{r\_max}(\frac{a_{r\_max}}{j})^2}{2} - \frac{j(\frac{a_{r\_max}}{j})^3}{6} \quad , \tag{4.34}$$

$$\frac{s_1}{2} = \frac{a_{r\_max}^3}{j^2} \quad , \tag{4.35}$$

$$a_{r\_max} = \sqrt[3]{\frac{s_1 j^2}{2}} \quad , \tag{4.36}$$

$$t_{ar\_max} = \frac{a_{r\_max}}{j} = \frac{\sqrt[3]{\frac{s_1 j^2}{2}}}{j} \quad . \tag{4.37}$$

Skupaj

$$t_{total} = 4t_{ar\_max} = 4\frac{a_{r\_max}}{j} = 4\sqrt[3]{\frac{s_1}{2j}} \quad . \tag{4.38}$$

Primer 2

Pri primeru 2 računamo čas, ki je potreben, da dvigalo opravi razdaljo s_2, ki je prekratka, da bi dvigalo razvilo svojo maksimalno hitrost v_{max}. Za izračun časa moramo najprej izračunati, kolikšna ta hitrost je v_{r_max}, ter nato iz nje izračunamo čas gibanja. Gibanje je sestavljeno iz prve, tretje, pete in sedme faze gibanja, pri čemer sta prva in tretja faza zopet ekvivalentni peti in sedmi fazi glede na čas in razdaljo.

$$t_{a\_max} = t_{a\_zero} = \frac{a_{max}}{j} \quad , \tag{4.39}$$

$$v_{a\_max} = \frac{a_{max}^2}{2j} \quad , \tag{4.40}$$

$$x_{a\_max} = \frac{a_{max}^3}{6j^2} \quad , \tag{4.41}$$

$$v_{r\_max} = v_{a\_max} + v_\delta + v_{a\_max} = 2v_{a\_max} + v_\delta \quad , \tag{4.42}$$

$$v_\delta = v_{r\_max} - 2v_{a\_max} \quad , \tag{4.43}$$

$$t_\delta = \frac{v_\delta}{a_{max}} \quad , \tag{4.44}$$

$$\begin{aligned}\frac{s_2}{2} =& x_{a\_max} + (v_{a\_max}t_\delta + \frac{a_{max}t_\delta^2}{2}) + \\ & ((v_{r\_max} - v_{a\_max})t_{a\_max} + \frac{a_{max}t_{a\_max}^2}{2} - \frac{jt_{a\_max}^3}{6}) \quad ,\end{aligned} \tag{4.45}$$

Po izrazu vseh količin enačbe 4.45 s pospeškom in trzajem pridemo do enačbe 4.46, kjer dobimo kvadratno neenačbo. Enačba ima dve rešitvi, eno negativno in eno pozitivno. V našem kontekstu hitrosti je pravilna le pozitivna.

$$\frac{s_2}{2} = \frac{a_{max}v_{max}}{2j} + \frac{v_{max}^2}{2a_{max}} \rightarrow s_2 = \frac{a_{max}v_{max}}{j} + \frac{v_{max}^2}{a_{max}} \quad , \tag{4.46}$$

$$v_{rmax} = \frac{a_{max}^2}{2j}(-1 + \sqrt[2]{1 + \frac{4s_2j^2}{a_{max}^3}}) \quad , \tag{4.47}$$

$$t_{jerk} = \frac{a_{max}}{j} \quad , \tag{4.48}$$

$$t_\delta = \frac{v_{max} - 2v_{a\_max}}{a_{max}} \quad , \tag{4.49}$$

$$t_{total} = 2(2t_{jerk} + t_\delta) \quad , \tag{4.50}$$

$$t_{total} = \frac{a_{max}}{j}\left(1 + \sqrt[2]{1 + \frac{4s_2j^2}{a_{max}^3}}\right) \quad . \tag{4.51}$$

Primer 3

Primer 3 nastopi takrat, ko je razdalja $s3$ dovolj velika, da dvigalo doseže svojo maksimalno hitrost.

$$t_{a\_max} = \frac{a_{max}}{j} \quad , \tag{4.52}$$

$$v_{a\_max} = \frac{a_{max}^2}{2j} \quad , \tag{4.53}$$

$$t_{v\_max} = \frac{v_{max} - 2v_{a\_max}}{a_{max}} = \frac{v - 2\left(\frac{a_{max}^2}{2j}\right)}{a_{max}} = \frac{v_{max}}{a_{max}} - \frac{a_{max}}{j} \quad , \tag{4.54}$$

$$x_{rest} = s3 - x_{limit2} \quad , \tag{4.55}$$

$$t_{x\_rest} = \frac{x_{rest}}{v_{max}} \quad , \tag{4.56}$$

$$t_{total} = 2(2t_{a\_max} + t_{v\_max}) + t_{x\_rest} \quad , \tag{4.57}$$

$$t_{total} = 2\left(2\frac{a_{max}}{j} + \left(\frac{v_{max}}{a_{max}} - \frac{a_{max}}{j}\right)\right) + \frac{s_3 - \left(\frac{v_{max}^2}{a_{max}} + \frac{v_{max}a_{max}}{j}\right)}{v_{max}} \quad , \tag{4.58}$$

$$t_{total} = \frac{a_{max}}{j} + \frac{v_{max}}{a_{max}} + \frac{s_3}{v_{max}} \quad . \tag{4.59}$$

4.2.2 Porabljena energija

Pri računanju energije, ki jo dvigalo porabi pri gibanju v eni smeri, smo uporabili osnovno enačbo za moč $P_{\#}$ ter iz nje izračunali energijo $E_{\#}$.

$$E_{\#} = \int P_{\#} dt \quad , \tag{4.60}$$

Zgornja in naslednje enačbe so splošno zapisane z znakom #, ki nadomešča x oziroma y.

Zaradi kompleksnih enačb smo se izognili integriranju le-teh s simulacijo premika. V simulacij smo čas razdelili na 10000 enako dolgih diskretnih intervalov ter za vsakega izračunali moč in energijo. Vsota vseh energij skozi 10000 delcev poti je bila zelo dober približek realno porabljene energije. Ker so takšne simulacije relativno zamudne, še posebno, če jih je potrebno izvesti mnogokrat, smo odločili za predhodno računanje energij. Pri predhodnem računanju energij smo za vse možne relativne premike dvigala v skladiščnem sistemu izračunali potrebno energijo preko simulacije in jo shranili kot serializiran objekt.

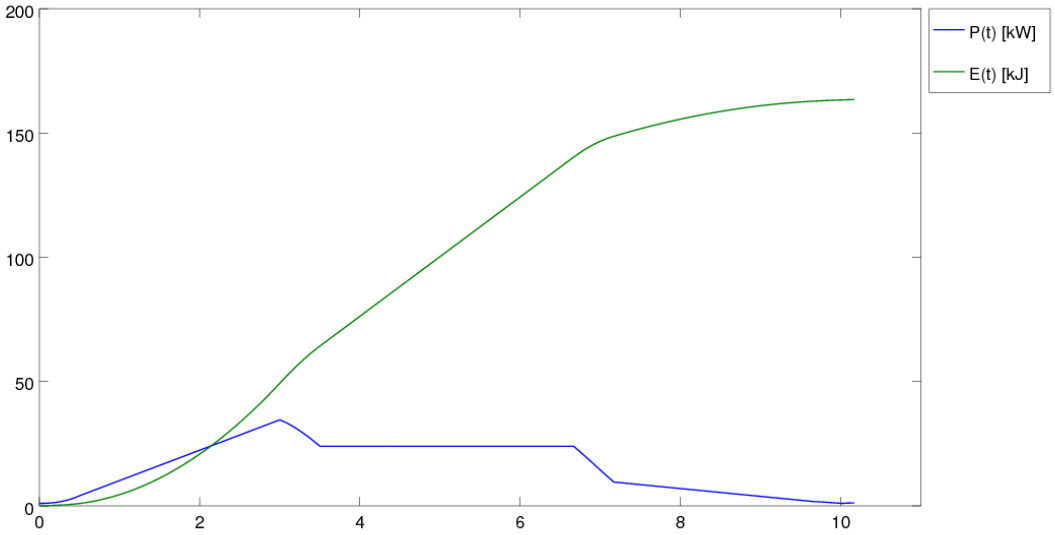
Ker se v končni fazi vse prenaša na motorje, kjer imamo rotacijsko gibanje, smo moč izračunali iz navorov. Celotni navor je sestavljen iz navora premikanja $M_T(t)$, navora vrtenja $M_R(t)$, navora trenja $M_F(t)$ in navora bremena $M_L(t)$. Pri tem smo v naših izračunih navor bremena zanemarili, ker je sama konstrukcija dvigala veliko težja od transportno skladiščnih enot.

$$P_{\#}(t) = M_{\#}(t)\omega_{\#}(t) \quad , \tag{4.61}$$

$$M_{\#}(t) = M_{T_{\#}}(t) + M_{R_{\#}}(t) + M_{F_{\#}}(t) + M_{L_{\#}}(t) \quad , \tag{4.62}$$

$$\omega_{\#}(t) = v_{\#}(t)\frac{2}{d_{\#}}i_{\#} \quad . \tag{4.63}$$

Na sliki 4.6 sta prikazani funkciji skupne energije $E(t)$ in moči $P(t)$ pri gibanju dvigala v horizontalni smeri za 10 m.



Slika 4.6: Graf skupne energije in moči v odvisnosti od časa

Računanje moči potrebne za horizontalno gibanja dvigala

$$M_{T_x}(t) = m_x a_x(t) \frac{d_x}{2} \frac{1}{i_x} \frac{1}{\eta_x} \quad , \tag{4.64}$$

$$M_{R_x}(t) = J_x \alpha_x(t) = J_x \frac{a_x(t)}{r_x} = J_x \frac{2a_x(t) i_x}{d_x} \quad , \tag{4.65}$$

$$M_{L_x}(t) = 0 \quad , \tag{4.66}$$

$$M_{F_x}(t) = k_x m_x g \frac{d_x}{2} \frac{1}{i_x} \frac{1}{\eta_x} \quad , \tag{4.67}$$

$$M_x(t) = m_x a_x(t) \frac{d_x}{2} \frac{1}{i_x} \frac{1}{\eta_x} + J_x \frac{2a_x(t) i_x}{d_x} + k_x m_x g \frac{d_x}{2} \frac{1}{i_x} \frac{1}{\eta_x} \quad , \tag{4.68}$$

$$P_x(t) = \frac{1}{\eta_x} m_x a_x(t) v_x(t) + \frac{J_x}{(\frac{d}{2i_x})^2} a_x(t) v_x(t) + \frac{1}{\eta_x} k_x m_x g v_x(t) \quad . \tag{4.69}$$

Računanje moči potrebne za vertikalno gibanje dvigala

$$M_{T_y}(t) = m_y a_y(t) \frac{d_y}{2} \frac{1}{i_y} + \frac{1}{\eta_y} \quad , \tag{4.70}$$

$$M_{R_y}(t) = J_y \alpha_y(t) = J_y \frac{2a_y(t) i_y}{d_y} \quad , \tag{4.71}$$

$$M_{L_y}(t) = m_y g \frac{d_y}{2} \frac{1}{i_y} \frac{1}{\eta_y} \quad , \tag{4.72}$$

$$M_{F_y}(t) = k_y m_y g \frac{d_y}{2} \frac{1}{i_y} \frac{1}{\eta_y} \quad , \tag{4.73}$$

$$M_y(t) = m_y a_y(t) \frac{d_y}{2} \frac{1}{i_y} + \frac{1}{\eta_y} + J_y \frac{2a_y(t) i_y}{d_y} + m_y g \frac{d_y}{2} \frac{1}{i_y} \frac{1}{\eta_y} + k_y m_y g \frac{d_y}{2} \frac{1}{i_y} \frac{1}{\eta_y} \quad , \tag{4.74}$$

$$P_y(t) = \frac{1}{\eta_y} m_y a_y(t) v_y(t) + \frac{J_y}{(\frac{d}{2i_y})^2} a_y v_x + \frac{1}{\eta_y} m_y g v_y(t) + \frac{1}{\eta_y} k_y m_y g v_y(t) \quad . \tag{4.75}$$

Skupna poraba energije celotnega sistema

Pri računanju skupne moči celotnega sistema za premik dvigala z ene lokacije na drugo smo simulirali tri različne pristope izmenjave presežne energije med pogoni.

Pristop 1
Pri prvem načinu smo simulirali pristop vezave elektromotorjev preko uporov. Pri tem pristopu se presežna energija sprosti na zavornih uporih, kjer od nje nimamo nobene koristi. V praksi je takšen pristop zelo pogost, saj je to najcenejši način izdelave sistema, ki pa ni učinkovit.

$$P(t) = P_{0inv} + P_{0aux} + \max(P_x(t), 0) + \max(P_y(t), 0) \quad . \tag{4.76}$$

Pristop 2
Pri drugem pristopu smo simulirali verzijo skladiščnega sistema, kjer si presežno energijo pogoni dvigala med sabo izmenjujejo. V tem primeru se energijska učinkovitost skladiščnega sistema najbolj izboljša v primerih, ko se dviganje v vertikalni smeri začne ob zaviranju v horizontalni smeri. Takšen pristop je nekakšen kompromis med pristopom 1 in pristopom 3.

$$P_{fx}(t) = \begin{cases} P_x(t) & \text{če } P_x(t) \geq 0 \\ \eta_{zw} P_x(t) & \text{drugače} \end{cases} \quad , \tag{4.77}$$

$$P_{fy}(t) = \begin{cases} P_y(t) & \text{če } P_y(t) \geq 0 \\ \eta_{zw} P_y(t) & \text{drugače} \end{cases} \quad , \tag{4.78}$$

$$P_{tmp}(t) = P_{0inv} + P_{0aux} + P_{fx}t + P_{fy}t \quad , \tag{4.79}$$

$$P(t) = \max(P_{tmp}, P_{0aux}) \quad . \tag{4.80}$$

Pristop 3

V tretjem pristopu smo simulirali skladiščni sistem, kjer se presežna energija vrne v omrežje, torej nam ponuja najbolj optimalno delovanje sistema. Vendar je v realnosti takšen pristop najdražji.

$$P_{fx}(t) = \begin{cases} P_x(t) & \text{če } P_x(t) \geq 0 \\ \eta_{zw} P_x(t) & \text{drugače} \end{cases} , \tag{4.81}$$

$$P_{fy}(t) = \begin{cases} P_y(t) & \text{če } P_y(t) \geq 0 \\ \eta_{zw} P_y(t) & \text{drugače} \end{cases} , \tag{4.82}$$

$$P(t) = P_{0inv} + P_{0aux} + P_{fx}t + P_{fy}t \quad . \tag{4.83}$$

4.3 Podatki

Podatki, nad katerimi smo opravljali optimizacijo skladiščnega sistema, so bila opravila dvigala, podana s strani sistema WMS. Posamezno opravilo je vsebovalo izvorno ter ponorno lokacijo transportno skladiščne enote. Na voljo smo imeli realna opravila, ki so bile zajeta z obratujočega skladiščnega sistema. Poleg realnih opravil pa smo izdelali umetni generator opravil, pri katerem smo imeli večji nadzor nad sistemom.

Posamezno opravilo skladiščnega sistema je predstavljeno z dvema skladiščnima lokacijama. Opravila skladiščnega sistema so lahko naslednjih tipov:

- prenos transportno skladiščne enote v skladiščni sistem,
- prenos transportno skladiščne enote iz skladiščnega sistem,
- prenos transportno skladiščne enote med dvema V/I mestoma,
- prenos transportno skladiščne enote med dvema skladiščnima mestoma.

Primer opravila skladiščnega sistema za prenos transportno skladiščne enote v skladišče je: T1302 $\rightarrow$ R2X27Y053, pri čemer:

- T1302 predstavlja V/I mesto skladiščnega sistema,
- R2X27Y053 predstavlja skladiščno mesto v skladiščnem sistemu.

4.3.1 Realna opravila, zajeta iz skladiščnega sistema

V času izdelave optimizacije smo imeli na razpolago 17377 zapisov opravil, ki jih je skladiščni sistem že izvedel. Poleg izvorne ter ponorne lokacije smo imeli pri posameznem opravilu polje *id*, ki se je z vsakim naslednjim opravilom povečeval. Iz opravil je bilo razvidno, da je bil naš skladiščni sistem takrat še relativno nov in še ni bil v polnem obratovanju. Opravila so namreč vsebovale veliko vmešavanj in neskladij, tako da smo morali ta opravila predhodno obdelati in izločiti določena opravila. Opravila smo pred optimizacijo obdelali po naslednjem postopku:

1. Vsa opravila smo uredili po njihovih *id*-jih, da smo dobili pravilno sekvenčno zaporedje. To zaporedje je bilo enako zaporedju, po katerem so bila ustvarjenje s strani sistema WMS.

2. Odstranili smo opravila, ki so imela izvorne lokacije enake ponornim lokacijam.

3. Odstranili smo opravila, ki so vsebovala premik skladiščno transportne enote z ene mize dvigala na drugo.

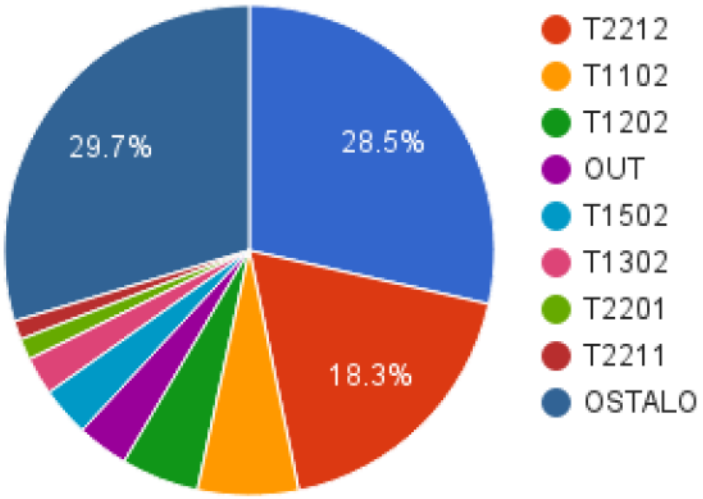
4. Opravila, ki so vsebovala premik transportno skladiščne enote s skladiščnega mesta na specifično mizo dvigala, smo preuredili. Preuredili smo jih tako, da smo jim nastavili pravilno ponorno lokacijo, ki smo jo ugotovili preko naslednjih opravil.

5. V primeru več enakih zaporednih opravil smo upoštevali le opravilo z največjim *id*-jem, ostala smo odstranili.

6. Določena opravila so imele ponorno ali izvorne lokacije označene z "OUT", kar ni veljavno skladiščno mesto. To so bila ročna posredovanja v skladiščni sistem. Takšna opravila smo si posebej označili in jih tudi ustrezno obravnavali pri sami optimizaciji.

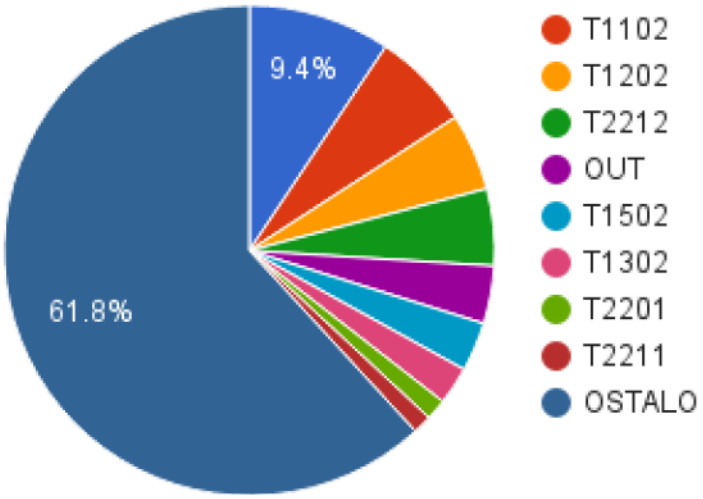
Skozi proces obdelave smo po zgornjem postopku zavrgli okoli 939 opravil (5,5 %). Nato smo nad vsemi preostalimi opravili opravili statistično analizo, da smo ugotovili, s kakšnimi opravili imamo opravka.

Na sliki 4.7 je prikazan tortni diagram, ki prikazuje delež osmih najpogostejših mest v skladišču, ki so bila uporabljena kot izvorne lokacije pri analiziranih opravilih. Vseh teh 8 lokacij skupno pokrije 70,3 % opravil, kar pomeni, da se je preostalih 7309 skladiščnih mest in 16 V/I mest uporabljalo le v 29,7 % opravil. Med temi osmimi mesti najbolj izstopata mesti $T2202$ in $T2212$, ki skupaj predstavljata začetek 46% vseh opravil.



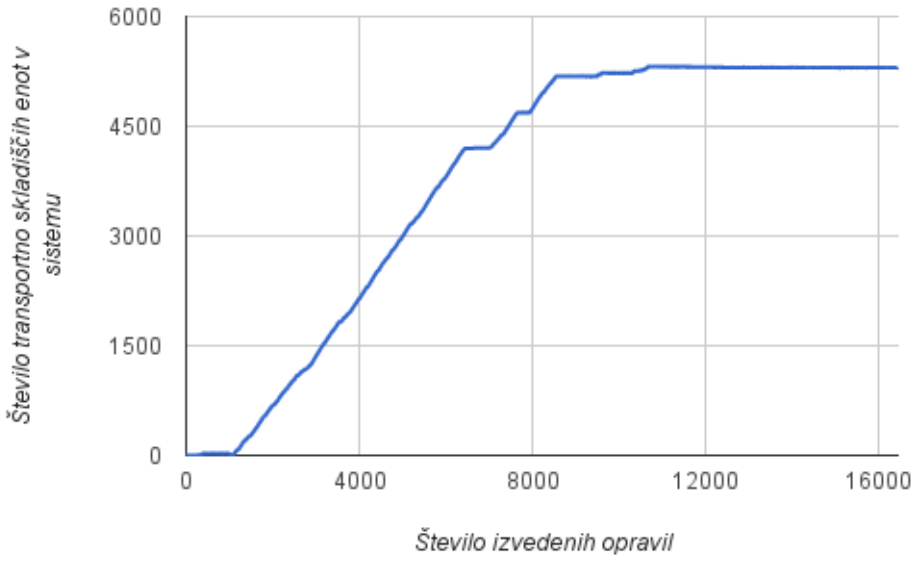
Slika 4.7: Tortni diagram najpogostejših začetnih mest

Druga analiza, ki smo jo opravili, je frekvenčna analiza končnih mest analiziranih opravil. Njeni rezultati najpogosteje uporabljenih mest kot končnih mest opravil so prikazani na sliki 4.8. V tem primeru so bila mesta uporabljena malo bolj enakomerno, vendar so še vedno štiri najpogostejša mesta pokrila 25,7 % vseh opravil.



Slika 4.8: Tortni diagram najpogostejših končnih mest

Slika 4.9 prikazuje število transportno skladiščnih enot v skladiščnem sistemu skozi opravila. Ker smo dobili vsa opravila iz skladiščnega sistema, ko transportno skladiščne enote še niso bilo vstavljene, se je pri polovici opravil le vstavljalo transportno skladiščne enote v skladišče. Število transportno skladiščnih enot v skladiščnem sistemu postane konstantno pri opravilu 8500, ko doseže 5143 enot v sistemu. To je 70 % vseh skladiščnih mest, ki so na voljo, torej je v sistemu ostalo 30 % skladiščnih mest, po katerih se lahko te transportno skladiščne enote poljubno premikajo.



Slika 4.9: Graf števila transportno skladiščnih enot skozi opravila

Poleg zgoraj naštetih analiz smo opravili še nekatere druge statistične analize. Iz njih smo ugotovili, da je bilo zelo veliko ročnih posegov v sistem. Poleg uporabe istih V/I skladiščnih mest pa se ravno tako skladiščna mesta niso uporabljala enakomerno, ampak so se nekatera izmed njih uporabljala pogosteje. Iz analize sklepamo, da se je naš skladiščni sistem takrat, ko so bili podatki zajeti, šele zaganjal, saj so polovico opravil predstavljala opravila uskladiščenja, s katerimi so se transportne skladiščne enote vstavljale v skladiščni sistem.

4.3.2 Generator opravil

Izdelave generatorja opravil smo se lotili po opravljeni analizi realnih opravil. Naš cilj je bil bolj enakomerno uporabiti V/I ter skladiščna mesta v sistemu. Želeli pa smo tudi imeti nadzor nad deležem opravil, ki jih lahko dvigalo opravi hkrati v enem koraku. Prepogosta uporaba istih skladiščnih mest skozi opravila namreč predstavlja velik problem pri optimizaciji. Ko pripelje dvigalo transportno skladiščno enoto na V/I mesto, mora le-to počakati določen čas, preden lahko to transportno skladiščno enoto pospravi nazaj v skladiščni sistem. Ravno tako pa se opravila z istimi mesti med seboj blokirajo in moramo pri njih paziti na to, katero opravilo bomo izvedli pred katerim. V primeru realnih opravil tako nimamo veliko optimizacijskega prostora, ker se opravila med seboj zelo blokirajo.
Izdelani generator podpira generiranje naslednjih tipov opravil:

- enojna opravila – z verjetnostjo P_S %
  - v skladiščni sistem: 50 %
  - iz skladiščnega sistema: 50 %
- dvojna opravil – z verjetnostjo $(100 - P_S)$ %
  - obe opravili v skladišče – 33 %, pri čemer sta V/I mesti izbrani naključno, skladiščni mesti pa sta sosednji skladiščni mesti v skladišču,
  - obe opravili iz skladišča – 33 %, pri čemer sta skladiščni mesti sosednji mesti v skladiščnem sistemu, V/I mesti pa sta izbrani naključno,
  - mešani opravili – 33 %
    - 1. opravilo ima izvor na naključnem skladiščnem mestu, ponor pa na naključnem V/I mestu skladišča,
    - 2. opravilo ima izvor na naključnem V/I mestu, ponor pa na sosednjem skladiščnem mestu 1. opravila.

Poglavje 5

Implementacija

Cilj naše optimizacije je bila optimalna razporeditev opravil za dvigalo z dvema mizama, pri samem algoritmu pa je bil poleg optimalne razporeditve zahtevan še kratek čas izvajanja. Algoritme smo namreč želeli izdelati tako, da bi se jih lahko poganjalo v realnem času, torej smo se omejili na največ 2 s časa izvajanja algoritma. Celotno optimizacijo smo implementirali v programskem jeziku Java.

5.1 Izgradnja grafa akcij

Večina algoritmov, ki smo jih implementirali, temelji na iskanju najkrajše poti v grafu. Tako smo izdelali metodo, ki z upoštevanjem vseh fizikalnih omejitev sistema izdela graf vseh možnih akcij. Graf, ki ga metoda izdela, temelji na stanjih, pri čemer posamezno vozlišče predstavlja določeno stanje skladiščnega sistema, povezava pa predstavlja akcijo, ki nas privede do tistega stanja. Določeno stanje je definirano z lokacijami vseh produktov, ki so v sistemu, ter na dvigalu. Poleg produktov hrani tudi vsa opravila, ki se trenutno izvajajo, ter tiste, ki se še morajo. Zaradi velike obsežnosti grafa se graf ne zgradi v enem koraku, ampak se skozi preiskovanje razvija po potrebi.

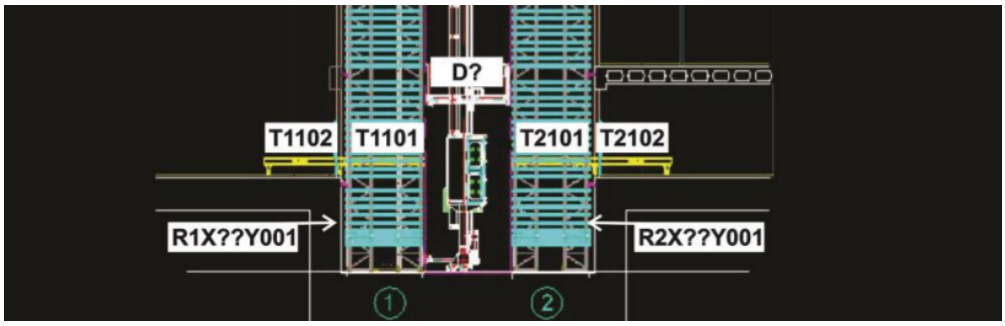
5.1.1 Fizikalne omejitve sistema

Pri izgradnji grafa smo upoštevali pravilno zaporedje izvajanja opravil, čas priprave skladiščnega sistema, čas in energijo za izvedbo določene akcije, omejitev dostopa dvigala ter možnosti izvedbe dvojnih opravil.

Pravilno zaporedje izvajanja opravil

Naš sistem lahko optimiziramo le v primeru mešanega izvajanja opravil. Pri tem pa moramo paziti, da ne preskočimo kakšno opravilo, ki je odvisno od opravila, ki ga želimo izvesti. Opravili sta med sabo odvisni v primeru, ko uporabljata isto izvorno ali pa ponorno mesto v skladiščnem sistemu. Pri opravilih, ki uporabljajo V/I transportna mesta za ponor ali izvor, pa moramo še dodatno paziti na posamezne mize V/I transportnega mesta, saj sta ti dve mizi na posameznem V/I transportnem mestu med sabo odvisni.

Odvisnost posameznih miz na V/I transportnem mestu je podrobno prikazana na sliki sheme skladiščnega sistema 5.1, kjer je razvidno, da je miza z oznako $T1102$ pred mizo z oznako $T1101$. Torej, če želimo oddati transportno skladiščno enoto na mizo $T1102$, mora biti miza $T1101$ predhodno prazna. Velja tudi obratno, tako da moramo opravila, ki uporabljajo isto V/I transportno mesto, izvršiti v enakem medsebojnem zaporedju, kot so bila ustvarjena s strani sistema WMS.



Slika 5.1: Shema miz na V/I trnasportnih mest

Čas čakanja, da se sistem pripravi

Pri odstranjevanju transportnih pladnjev iz sistema preko določenega V/I transportnega mesta moramo počakati določen čas t_{change}, preden se ta transportni pladenj dejansko odstrani. Ravno tako moramo vedno, ko damo transportni pladenj na V/I transportno mesto, počakati vsaj $t_{process}$ časa, preden ga lahko odnesemo nazaj v skladiščni sistem.

Čas izvajanja in porabljena energija

Pri izračunavanju cen povezav grafa, na podlagi katerih so potekale optimizacije, smo uporabili natančne izračune časa in energije. Pri času smo upoštevali čas gibanja dvigala, čas izvajanja akcije ter čas morebitnega čakanja na pripravo skladiščnega sistema. Pri energiji pa smo upoštevali energijo, ki jo porabi sistem za delovanje med samim premikanjem dvigala.

Omejitev dostopa do skladiščnih mest

Zaradi fizične strukture našega skladiščnega sistema z obema mizama na dvigalu ni bilo možno dostopati do vseh skladiščnih mest. Desna miza na dvigalu ni mogla dostopati do skrajno levih mest v skaldiščnem sistemu in obratno. Tako smo morali za vsako opravilo preveriti, ali lahko posamezna miza na dvigalu dostopa tako do izvorne kot do ponorne lokacije opravila.

Dvojna izvedba opravil

Ker smo izdelovali graf vseh možnih akcij, smo morali upoštevati tudi možnost akcije na obeh mizah dvigala hkrati. Pri takšni akciji dosežemo veliko optimizacijo sistema, saj lahko nalaganje transportnega pladnja na mizo dvigala ali pa razlaganje poteka hkrati na obeh mizah. Vendar pa je razpoznavanje takšnih akcij časovno kompleksno $O(r)$, pri čemer je r število preostalih opravil v trenutnem stanju skladiščnega sistema. Sicer je to linearna časovna zahtevnost, vendar takšno razpoznavanje dvojnih akcij poteka v vsakem vozlišču grafa posebej, kar upočasni samo izgradnjo grafa.

5.1.2 Graf

Zaradi vseh predhodno omenjenih omejitev je bil graf zelo razvejan in zelo podoben drevesni strukturi. Največja omejitev, ki je k temu pripomogla, je bil čas, ki mora preteči, da skladišče pripravi V/I transportno mesto. Število vseh stanj oziroma vozlišč v grafu eksponentno narašča po enačbi:

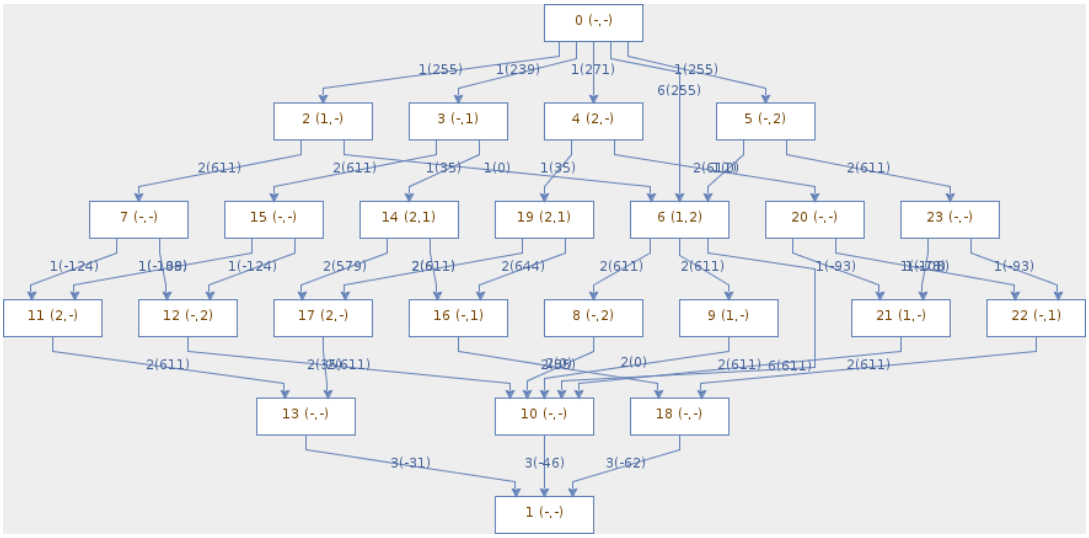
$$num\_vert(n) = 2n + 2na(n-1) \quad , \tag{5.1}$$

$$a(n) = 1 + 5n + 4na(n-1) \quad . \tag{5.2}$$

Pri tem n predstavlja število opravil, na podlagi katerih gradimo graf, $a(n)$ pa je pomožna funkcija.

Na sliki 5.2 je prikazan graf vseh možnih akcij, ki je ustvarjen iz opravil:

- 1 R1X10Y022 R1X22Y089
- 2 R1X11Y022 R1X23Y089



Slika 5.2: Graf akcij dveh neodvisnih opravil

Prikazani opravili na grafu sta med sabo neodvisni in imata sosednji skladiščni mesti za izvorni ter tudi ponorni lokaciji opravil. Ker sta opravili sestavljeni le iz skladiščnih mest in ne tudi V/I transportnih mest, je graf veliko manjši, kot bi bil v nasprotnem primeru.

Vozlišča na grafu so označena z oznakami $A(B, C)$, pri čemer parameter A predstavlja sekvenčno zaporedje vozlišča, B *id* opravila, ki jo trenutno izvaja dvigalo z levo mizo dvigala, in C *id* opravila, ki jo dvigalo trenutno izvaja z desno mizo dvigala.

Povezave na grafu pa so označene z $A(B)$, pri čemer parameter A predstavlja tip akcije, s katero pridemo v naslednje stanje, parameter B pa potreben čas za izvedbo te akcije. Tipi akcij so: 0 – začetek, 1 – nalaganje, 2 – razlaganje, 3 – konec in 6 – dvojna akcija.

5.2 Dinamično izvajanje opravil

Pri analiziranju algoritmov smo implementirali možnost dinamičnega izvajanja opravil. Pri dinamičnem izvajanju opravila, ki jih želimo optimizirati, optimizacijskim algoritmom dodajamo sproti in ne vseh naenkrat. Namen dinamičnega dodajanja opravil je bil čim bolj realno simulirati dogajanje v pravem skladiščnem sistemu. Naše dinamično izvajanje nadziramo z dvema parametroma:

- *Število opravil na iteraciji* določa, koliko bo vseh opravil v posamezni iteraciji optimizacije.

- *Število ohranjenih opravil* predstavlja število opravil, ki se prenese v naslednjo iteracijo optimizacije.

V nadaljevanju za označevanje parametrov dinamičnega programiranja uporabljamo notacijo $A(B)$, pri čemer A predstavlja število opravil na iteracijo, B pa število ohranjenih opravil. Primer: dinamično izvajanje opravil 10(6) pomeni, da bodo algoritmi v vsaki iteraciji optimizirali 10 opravil. Ko bodo vrnili optimalno pot, se bodo pričele izvajati opravila, ki jih algoritmi vrnejo. Nato, ko bo ostalo samo še 6 nerazrešenih opravil, se bodo dodala preostala, tako da bo skupno zopet 10 opravil. Ta opravila bodo spet podana algoritmom za optimiziranje in tako se bo postopek ponavljal, vse dokler ne optimiziramo vseh opravil.

5.3 Staranje opravil

Po sami implementaciji dinamičnega izvajanja opravil smo morali implementirati še staranje opravil. Z implementacijo staranja opravil smo želeli preprečiti, da bi se kakšno opravilo predolgo zadrževala nerazrešena v skladiščnem sistemu. Tako vsakemu prejetemu opravilu dodelimo zaporedno številko, na podlagi katere določamo prioriteto opravila.

5.4 Algoritmi

Pri implementaciji večine algoritmov je bilo potrebno za njihovo optimalno delovanje ustrezno nastaviti njihove parametre. Posebnih parametrov nismo imeli le pri Dijkstrovem ter Bellman-Fordovem algoritmu, ki sta splošna algoritma za iskanje najkrajše poti v grafu. Pri ostalih algoritmih pa smo parametre nastavili v odvisnosti od posameznega problema, ki ga optimiziramo:

- vrsta optimizacije (časa ali energije),
- tip skladiščnega sistema (uporaba ene ali dveh miz na dvigalu).

5.4.1 Genetski algoritem

Pri genetskem algoritmu smo uporabili eno točkovno križanje, pri mutaciji pa smo z naključnega vozlišča izbrali drugo akcijo, kot je bila predhodno izbrana. Osnovne parametre algoritma smo imeli nastavljene na:

- velikost populacije: 32,
- število iteracij: 100,
- število potomcev na iteracijo: 8,
- verjetnost mutacije: 0,2 %.

Vse vrednosti so za genetski algoritem in razsežnosti našega problema zelo nizke, razlog za to pa je predvsem časovna omejitev na dve sekundi optimizacije, ki smo jo imeli.

Poleg osnovnih parametrov pa smo imeli še ustrezno nastavljena parametra za staranje, na podlagi katerih smo določali višjo prioriteto starejšim opravilom:

- spodnja starostna meja opravil: 20,
- zgornja starostna meja opravil: 40.

5.4.2 Optimizacija s kolonijami mravelj

Pri optimizaciji s kolonijami mravelj so bili ravno tako kot pri genetskih algoritmih parametri nastavljeni na zelo nizke vrednosti. Te vrednosti so bile:

- število korakov na iteracijo: 5,
- število iteracij: 100,
- število mravelj: 20,
- število mravelj na iteracijo: 1,
- območje feromonov: $[0 \rightarrow 100]$,
- izhlapevanje feromonov na iteracijo: 1,
- verjetnost izbire poti s feromoni: 0,8 %.

Za določitev prioritete starejšim opravilom smo uporabili enake parametre kot pri genetskem algoritmu:

- spodnja starostna meja opravil: 20,
- zgornja starostna meja opravil: 40.

5.4.3 Algoritem A*

Pri algoritmu A* smo uporabili več različnih hevrističnih funkcij v odvisnosti od posameznega optimizacijskega problema. Hevristična funkcija 5.3 je v osnovi sestavljena iz preostanka opravil, ki jih moramo še opraviti, ter opravil, ki jih izvajamo v danem trenutku. Pri preostanku opravil upoštevamo čas, ki je potreben za izvedbo opravil v sekvenčnem zaporedju z uporabo le ene mize na dvigalu. Pri upoštevanju opravil, ki se izvajajo v tem trenutku, pa kaznujemo naslednje akcije, ki bi povzročile, da bi bili na dvigalu dve opravili s ponoroma na različnih koncih skladišča, ter nagradimo dvojne akcije.

$$\begin{aligned} h(x) = remaining\_tasks * 1 + time(remaining\_tasks) * 1.5 + \\ waiting\_time * 1 + h_1(x) - h_2(x) + h_3(x) \quad , \end{aligned} \tag{5.3}$$

$$h_1(x) = \begin{cases} 6, & \text{if } taken\_places = 1 \\ 12, & \text{if } taken\_places = 2 \\ 0, & \text{otherwise} \end{cases} , \tag{5.4}$$

$$h_2(x) = \begin{cases} 12, & \text{if } double\_action \\ 0, & \text{otherwise} \end{cases} , \tag{5.5}$$

$$h_3(x) = \begin{cases} 6, & \text{if } opposite\_direction \\ 0, & \text{otherwise} \end{cases} . \tag{5.6}$$

Pri tem so njeni parametri:

- h: hevristična funkcija,
- x: neko stanje skladiščnega sistema (vozlišče v grafu),
- $taken\_places$: število zasedenih miz na dvigalu v trenutnem stanju,
- $remaining\_tasks$: preostanek opravil, ki jih moramo še izvesti,

- *time*: čas, ki je potreben za izvedbo posameznega opravila ($nalaganje + premik(start \rightarrow end) + razlaganje$),
- *waiting_time*: čas čakanja, da skladišče pripravi V/I transport, preden lahko nad njim izvršimo naslednjo akcijo,
- *double_action*: ali pridemo v trenutno stanje x preko dvojne akcije,
- *opposite_direction*: novo opravilo je v drugi smeri kot trenutna pot.

V primeru energijske optimizacije so bili parametri rahlo prilagojeni, predvsem pa smo časovne vrednosti nadomestili z energijskimi. Prioriteta opravil glede na njihovo starost pa se je upoštevala že v koraku pred računanjem hevristične funkcije na podoben način, kakor se je pri genetskem algoritmu in optimizaciji s kolonijami mravelj.

5.4.4 Soseščina

Optimiziranje razporeditve opravil z algoritmom soseščine temelji na odločitvenem pristopu in ne preiskovalnem kakor ostali algoritmi, ki smo ji implementirali. Zanj smo se odločili, ko smo razmišljali, kako bi se optimizacije takšnega problema lotili ročno, brez uporabe računalnika. Na začetku algoritem soseščine vsa opravila, ki jih lahko v danem trenutku izvede, shrani v podatkovno strukturo KD-drevo na podlagi izvornih mest. Opravila, ki jih v danem trenutku še ne moremo izvesti zaradi odvisnosti od ostalih opravil, pa se bodo dodajale v podatkovno strukturo KD-drevo skozi čas, ko se bodo opravila, od katerih so odvisna, že razrešila. V danem trenutku se akcija, ki jo bo skladiščni sistem izvedel v naslednjem trenutku, določi na podlagi stanja dvigala. Ločimo naslednje primere:

- *0 zasedenih miz na dvigalu*: V tem primeru ni nobenega tekočega opravila, tako se bo v naslednjem trenutku pričelo izvajati novo opravilo. Kandidati za novo opravilo so le opravila, ki imajo izvorno lokacijo najbližje trenutni lokaciji dvigala v skladiščnem sistemu.

- *1 zasedena miza na dvigalu*: V primeru, ko ima dvigalo zasedeno eno mizo, pomeni, da že izvaja eno opravilo. Naslednja akcija bo bodisi začetek novega opravila ali pa zaključitev trenutnega opravila. Morebitno novo opravilo, ki bi se lahko začelo, se izbira le na podlagi opravil, ki imajo izvorno lokacijo v okolici poti, ki se prične na trenutni lokaciji dvigala in zaključi na ponorni lokaciji trenutnega opravila, ki se že izvaja.

- *2 zasedeni mizi na dvigalu*: V tem primeru dvigalo že izvaja dve opravili, tako da sta kandidata za naslednji akciji le zaključitvi trenutno tekočih opravil.

Pri iskanju dani lokaciji najbližjih opravil smo pričeli z opravili v radiju enega metra od trenutne lokacije dvigala. Ta radij smo iterativno povečevali, dokler nismo našli vsaj 5 najbližjih opravil oziroma; v primeru, da smo imeli vseh opravil manj kot 5, pa smo upoštevali vsa opravila pri odločitvenem procesu. V primeru iskanja najbližjih opravil v okolici določene poti smo celotni poti določili 20 enako oddaljenih točk in v okolici teh poiskali opravila z najbližjimi izvornimi lokacijami. Med posameznimi opravili smo se odločali na podlagi naslednjih parametrov:

- razdalja izvora in ponora opravila do začrtane poti že tekočega opravila oziroma točke dvigala,
- starost opravila,
- cena opravila (čas oziroma energija),
- čas priprave izvorne in ponorne lokacije opravila.

Poleg zgornjih parametrov smo implementirali še dodatno preverjanje dvojnih opravil, ki smo jim dodelili dodatno prioriteto. Pri preverjanju dvojnih opravil smo upoštevali tudi opravila, ki jih trenutno še ne moremo izvesti, ker so odvisna od še ne izvedenih opravil. V ta namen smo implementirali poseben parameter, s katerim smo nadzirali, ali takšna opravila glede na njihovo starost počakamo ali ne.

Ko se določena akcija izbere in označi kot izvedena, se v primeru, da je bila akcija zaključek trenutnega opravila, opravilo odstrani iz trenutno tekočih opravil, dvigalo pa premakne na ponorno lokacijo tega opravila. Poleg tega dodamo v KD-drevo vsa opravila, ki so bila odvisne samo še od trenutno zaključenega opravil. V primeru, da začnemo izvajati novo opravilo, dvigalo pomaknemo na izvorno lokacijo tega opravila in opravilo odstranimo iz KD-drevesa. Nato postopek z izbiro naslednje akcije ponovimo, vse dokler ne zaključimo vseh opravil.

Prednost takšnega pristopa optimizacije razporeditve opravil je predvsem hitrost izvajanja. Časovna zahtevnost takšnega pristopa je $O(n \log(n))$, pri čemer je n število opravil, ki jih moramo optimizirati. To pa lahko v našem primeru še izboljšamo na $O(n)$ z uporabe fiksnih mej v podatkovni strukturi KD-drevo.

Poglavje 6

Rezultati

Optimizacijo skladiščnega sistema smo opravili v dveh korakih. V prvem smo primerjali vse implementirane algoritme med sabo pri enakih karakteristikah skladiščnega sistema. V drugem koraku pa smo izvedli še optimizacijo razporeditve opravil pri različnih karakteristikah skladiščnega sistema za algoritem, ki se je v prvem delu analiziranja odrezal najbolje.

Vse optimizacije smo izvedli nad realnimi in umetno ustvarjenimi opravili, njihove rezultate pa smo primerjali s sekvenčno izvedbo opravil pri uporabi le ene mize na dvigalu. Pri rezultatih, ki smo jih dobili skozi analizo na realnih podatkih, ki smo jih imeli na voljo v času analiziranja, sta nas presenetila predvsem čas izvajanja Dijkstrovega algoritma ter majhno izboljšanje časa oziroma energije. Po analiziranju realnih opravil smo ugotovili, da je bil razlog za slabe rezultate relativno nov skladiščni sistem, ki takrat še ni prišel v fazo normalnega obratovanja. Težava je bila namreč v tem, da so se zelo pogosto uporabljala ista V/I transportna mesta in tudi skladiščna mesta. Tako smo imeli zelo malo optimizacijskega prostora, ker so bila opravila med seboj odvisna in optimizacijski algoritmi niso prišli do izraza. Na podlagi tega smo se odločili, da v nadaljevanju predstavimo rezultate, ki smo jih pridobili z analizo nad umetno ustvarjenimi opravili.

6.1 Primerjava algoritmov

Za primerjavo uspešnosti posameznih algoritmov, smo izvedli optimizacijo statičnega in dinamičnega izvajanje opravil. Naš cilj je bil najprej pri statični optimizaciji prilagoditi parametre algoritmov, da nam bodo ti vračali rešitve za razporeditve petdesetih opravil v časovnem območju dveh sekund. Nato pa smo izvedli še dinamično analizo, ki je boljša simulacija realnega delovanja skladiščnega sistema.

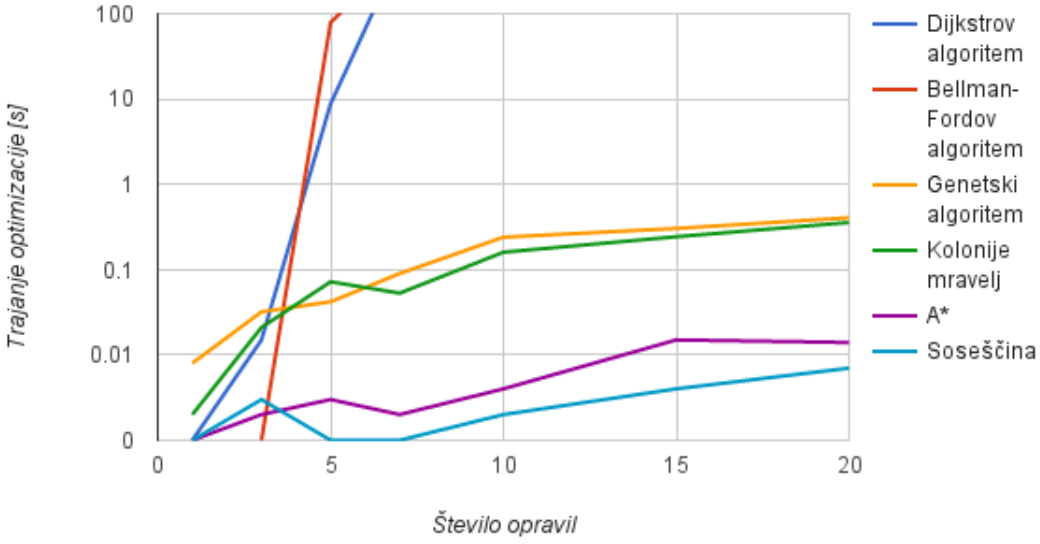
6.1.1 Statično izvajanje opravil

Pri statični optimizaciji razporeditve opravil smo naenkrat dali cel sklop opravil posameznemu algoritmu za optimizacijo. Ko je algoritem opravil optimizacijo, smo izračunali čas in energijo, ki bi jo pri tem porabil naš skladiščni sistem. Rezultate smo primerjali s sekvenčno izvedbo opravil v takšnem zaporedju, kot so bila ustvarjena s strani obstoječega sistema WMS oziroma našega generatorja opravil. Pri opravljenih statičnih optimizacijah nismo upoštevali staranja opravil, ker v tem primeru ni bilo relevantno.

Sklopi opravil so bili veliki od enega pa do petsto opravil. Pri tem smo posameznim algoritmom avtomatično prekinili izvajanje v primerih, ko so za optimizacijo porabili več kot nekaj minut, in večjih sklopov opravil nad njimi nismo več poskušali optimizirati. Rezultati, opisani v nadaljevanju, so pridobljeni pri optimizaciji umetno ustvarjenih opravil, pri čemer smo imeli nastavljeno verjetnost dvojnih opravil na 50 %. Pri vezavi elektromotorjev za premikanje dvigala smo upoštevali prvi pristop, pri čemer upoštevamo, da so elektromotorji vezani preko uporov, torej nimamo nobenega izkoristka odvečne energije.

Časovna optimizacija

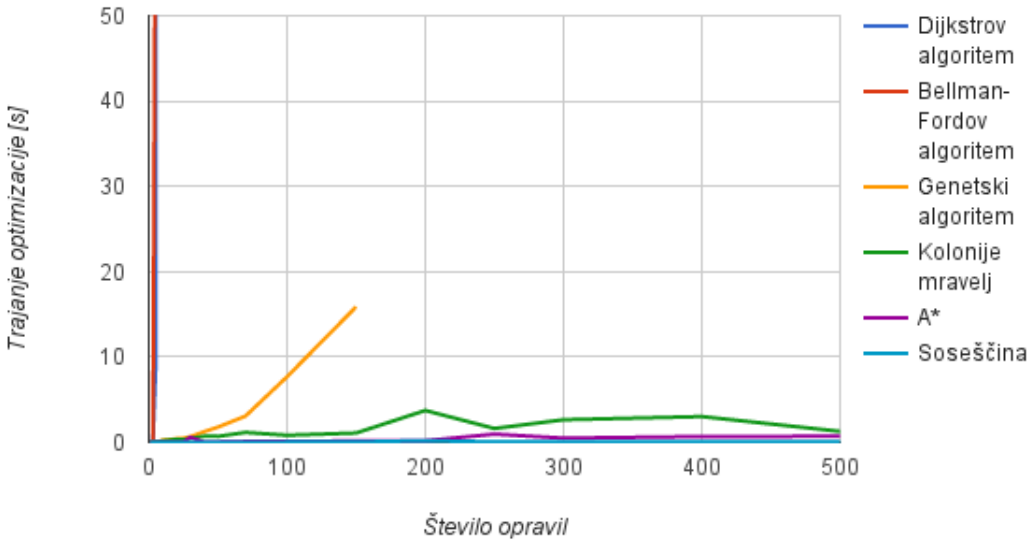
Pri časovni optimizacij sta bila Dijkstrov in Bellamn-Fordov algoritem najpočasnejša. Naše časovne zahteve sta prekoračila že pri optimiziranju razporeditve petih opravil. Ostali algoritmi pa so imeli do razporeditve dvajsetih opravil (slika 6.1) zelo podobne rezultate.



Slika 6.1: Časovna optimizacija – trajanje (20 opravil)

Pri optimizaciji sklopov opravil, velikih nad dvajset opravil (slika 6.2), sta po času optimizacije začela izstopati genetski algoritem ter optimizacija s kolonijami mravelj. Čas optimiziranja je pri genetskem algoritmu začel linearno naraščati po pričakovanjih, pri optimizaciji s kolonijami mravevlj pa se ta čas nekaj časa naraščal, nato pa začel padati. Razlog za takšno obnašanje je bila njegova definicija, kjer je njegovo trajanje pogojeno s številom mravelj, številom iteracij ter številom korakov v posamezni iteraciji. Tako so mravlje pri manjšem grafu večkrat našle pot od izvora do cilja in je bilo potrebno večkrat posodobiti poti. Pri večjem številu opravil pa je bil graf veliko večji in so tako mravlje manjkrat prišle od izvora do cilja.

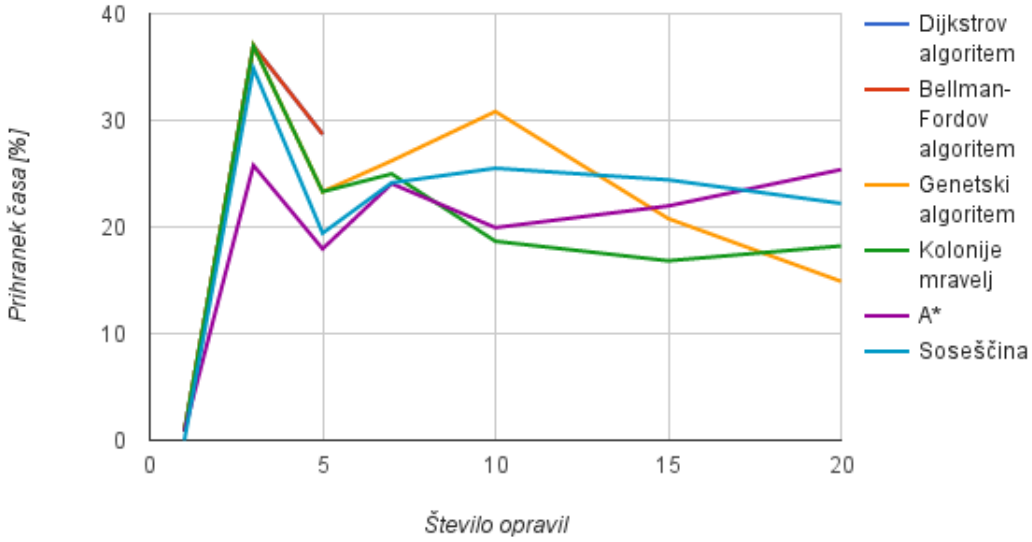
Glede časa izvajanja posamezne optimizacije sta se najbolje odrezala algoritem A* in algoritem soseščine. Algoritem A* je optimiziral posamezne sklope opravil v območju 1 sekunde, algoritem soseščine pa v območju 0,1 sekunde.



Slika 6.2: Časovna optimizacija – trajanje (500 opravil)

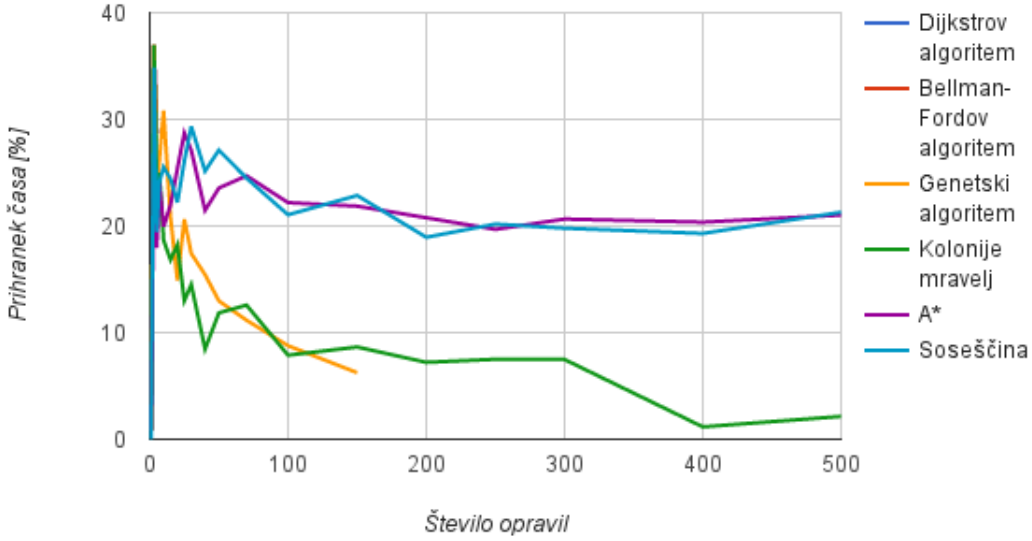
Rezultati, ki so jih algoritmi dosegli pri svojih optimizacijah, so bili najboljši pri Dijsktrovem in Belman-Fordovem algoritmu. To smo tudi pričakovali, saj oba algoritma zagotavljata optimalno rešitev problema. Pri optimiziranju treh opravil sta dosegla 36,98 % pohitritev, pri optimiziranju petih opravil pa 28,7 % pohitritev izvedbe opravil v primerjavi s sekvenčno izvedbo. Tem rezultatom so se približali tudi ostali analizirani algoritmi.

Pri optimiziranju do dvajset opravil (slika 6.3), med posameznimi algoritmi ni bilo velikega odstopanja glede pohitritve, ki so jo dosegli. V povprečju so vsi algoritmi dosegli okoli 20 % pohitritev v primerjavi s sekvenčno izvedbo opravil.



Slika 6.3: Časovna optimizacija – pohitritev (20 opravil)

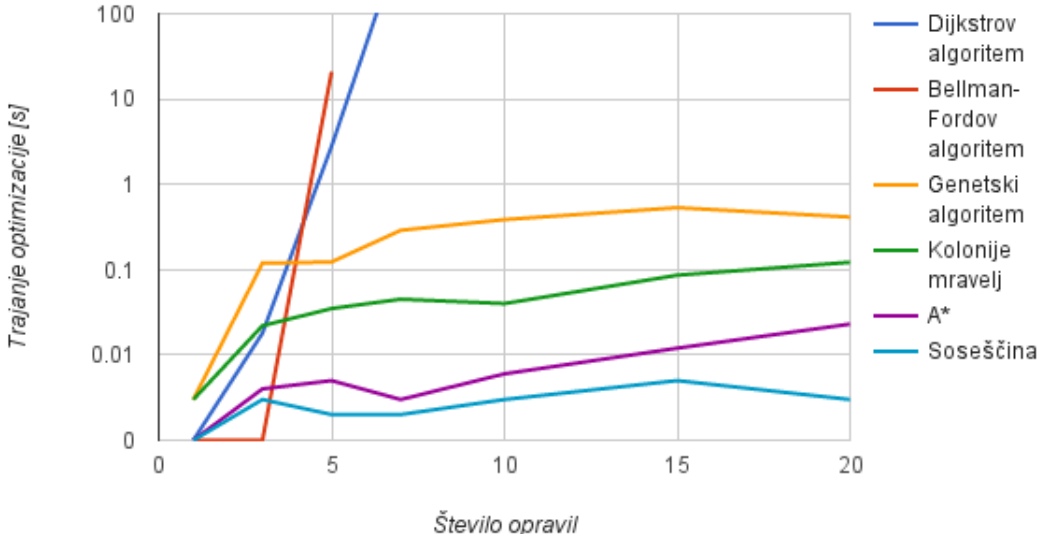
Pri sklopih opravil, večjih od dvajset opravil (slika 6.4), sta se genetski algoritem ter optimizacija s kolonijami mravelj pričela slabšati. Njuni rezultati so zelo hitro padli pod 10 % pohitritev, saj zaradi časovne omejitve nista mogla preiskati dovolj prostora. Algoritem A* in algoritem soseščine pa sta tudi pri večjih sklopih opravil ohranila 20 % pohitritev.



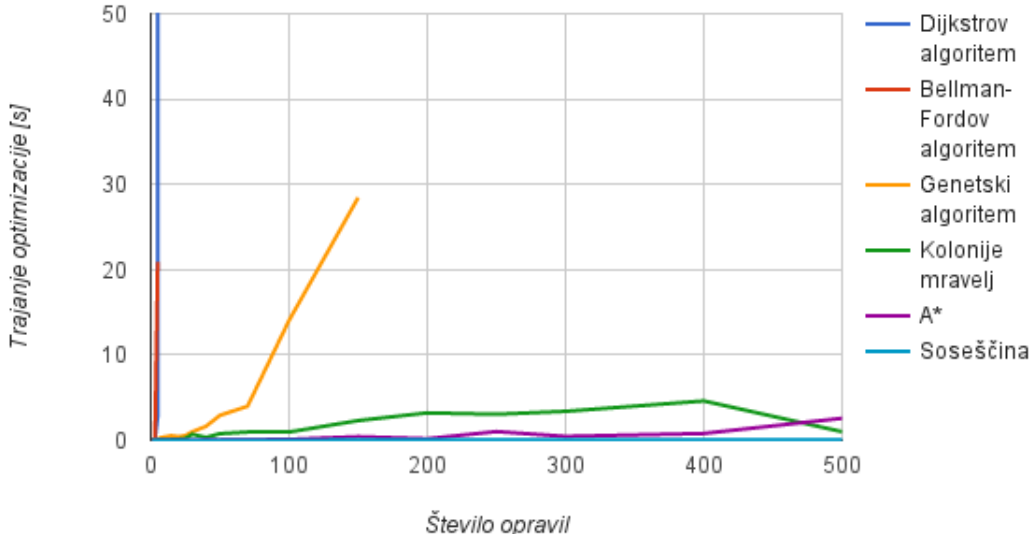
Slika 6.4: Časovna optimizacija – trajanje (20 opravil)

Energijska optimizacija

Pri energijski optimizaciji so bili časi izvedb optimizacij posameznih algoritmov (slika 6.5 in slika 6.5), zelo podobni časom časovne optimizacije. Najpočasnejša sta bila zopet Dijkstrov ter Bellman-Fordov algoritem, najhitrejša pa algoritem A* in algoritem soseščine.

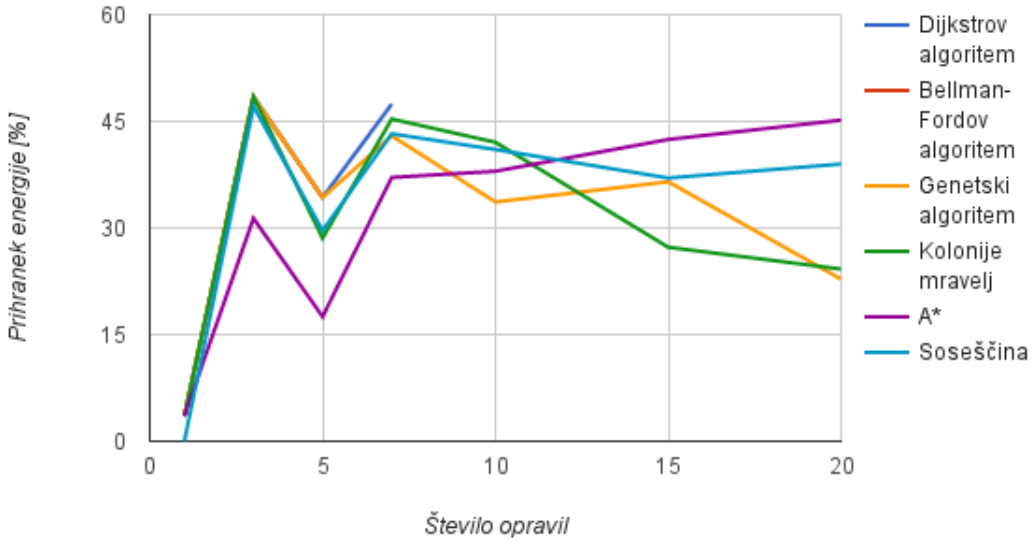


Slika 6.5: Energijska optimizacija – trajanje (20 opravil)



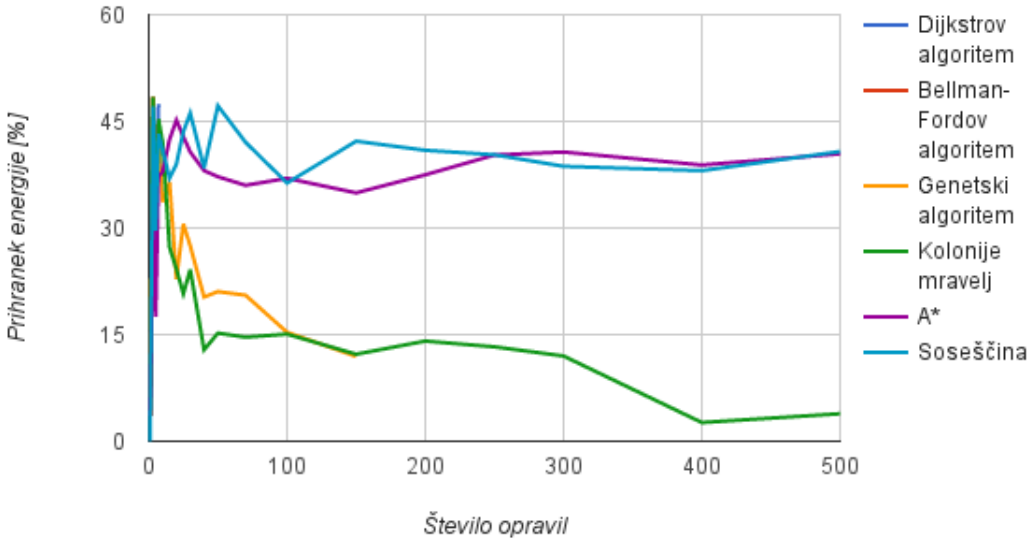
Slika 6.6: Energijska optimizacija – trajanje (500 opravil)

Pri optimiziranju do dvajset opravil (slika 6.7), med posameznimi algoritmi ni bilo velikega odstopanja glede na optimizacijo porabe energije. Dijkstrov in Bellman-Fordov algoritem sta pri optimiziranju porabe energije treh opravil dosegla 48,44 % optimizacijo in pri optimiziranju petih opravilih 34,29 % optimizacijo energije. Tem rezultatom so se zelo približali tudi ostali algoritmi, odstopal je le algoritem A*, ki je dosegel okoli 10 % slabše rezultate. Pri optimiziranju desetih opravil so imeli genetski algoritem, optimizacija s kolonijami mravelj, algoritem A* in algoritem soseščine zelo podobne optimizacijske rezultatem in sicer okoli 40 % optimizacijo. Pri večjih sklopih opravil pa so se pojavila odstopanja med posameznimi algoritmi.



Slika 6.7: Energijska optimizacija – optimiziranje (20 opravil)

Pri večjih sklopih opravil (slika 6.8) sta se genetski algoritem in optimizacija s kolonijami mravelj tako kot pri časovni kot tudi pri energijski optimizaciji začela slabšati. Pri optimiziranju stotih opravil sta oba uspela energijsko optimizirati opravila za le še okoli 15 % vrednosti sekvenčne izvedbe opravil. Pri večjih sklopih pa je bila dosežena optimizacija še manjša. Medtem je algoritem A* z večanjem sklopov opravil uspel izboljšati optimizacijo na 40 % in se s tem približal algoritmu soseščine. Pri algoritmu soseščine pa število opravil v posameznem sklopu ni pretirano vplivalo na njegov rezultate in je tako večino časa ohranjal 40 % optimizacijo energije v primerjavi s sekvenčno izvedbo vseh opravil.



Slika 6.8: Energijska optimizacija – optimiziranje (500 opravil)

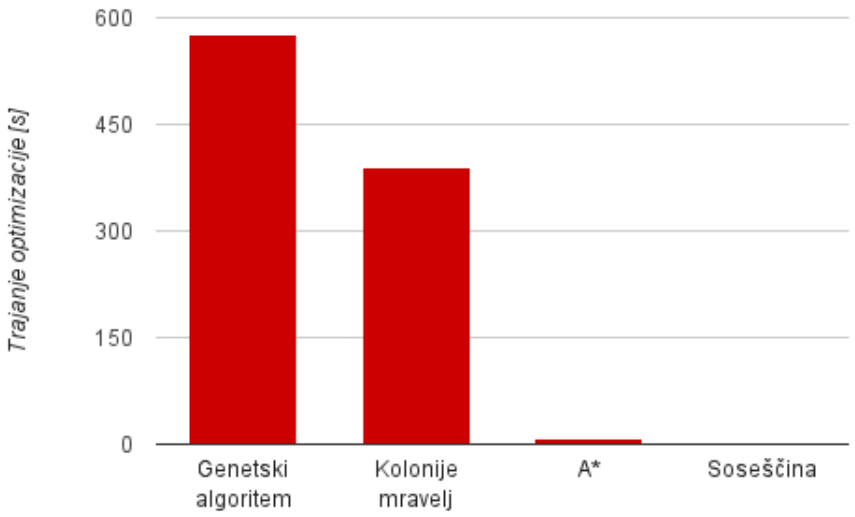
6.1.2 Dinamično izvajanje opravil

Pri dinamičnem izvajanju opravil smo dinamično opitmizirali 500 opravil, pri tem pa uporabili konfiguracijo 20(19). To pomeni, da so algoritmi imeli ves čas 20 opravil, ki so jih morali optimizirati. Zatem, ko smo na podlagi rešitve posameznega optimizacijskega algoritma simulirali eno opravilo, smo k preostalim opravilom dodali eno novo opravilo in vsa opravila znova podali optimizacijskemu algoritmu. Postopek smo ponavljali, dokler nismo razrešili vseh 500 opravil. Tako so morali vsi algoritmi sklop dvajsetih optimizirati okoli petstokrat. To moramo upoštevati tudi pri končnem skupnem času, ki so ga posamezni algoritmi potrebovali za dinamično optimiziranje petstotih opravil. Ker je bila naša zahteva opravila optimizirati v dveh sekundah, so morali za uspešno izpolnitev te zahteve algoritmi izvesti dinamično optimizacijo petstotih opravil v največ 1000 sekundah.

Pri analizi dinamičnega izvajanja algoritmov smo izpustili analizo Dijkstrovega in Bellman-Fordovega algoritma, ker je bil sklop dvajsetih opravil veliko preobsežen za njuno optimizacijo. Pri sami optimizaciji smo upoštevali tudi staranje opravil, tako da so starejša opravila v optimizacijskih algoritmih imela pri odločanju med posameznimi opravili večjo prioriteto. Pri analiziranju smo kakor pri statičnem izvajanju opravil tudi v tem primeru uporabili umetno ustvarjena opravila pri 50 % verjetnosti dvojnih opravil. Pri vezavi elektromotorjev za premikanje dvigala pa smo zopet upoštevali, da so elektromotorji vezani preko uporov.

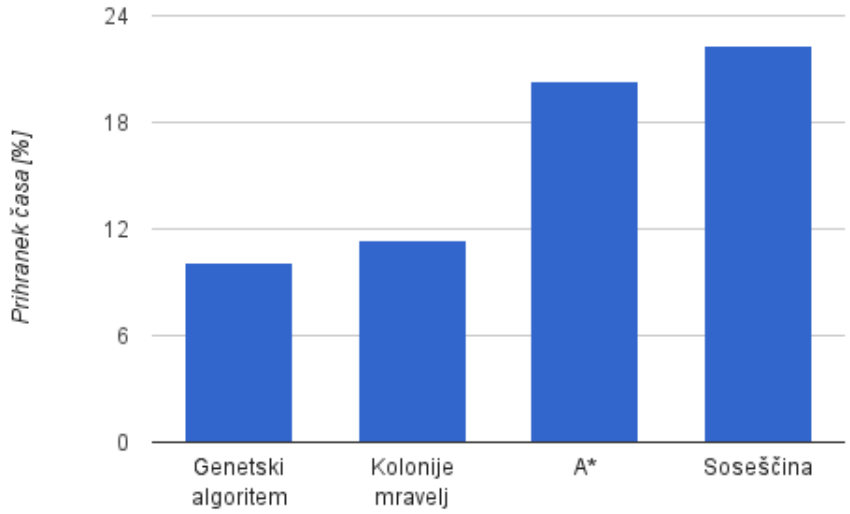
Časovna optimizacija

Pri časovni optimizaciji (slika 6.9), sta bila najpočasnejša genetski algoritem ter optimizacija s kolonijami mravelj. Veliko hitrejša od njiju sta bila algoritem A* ter algoritem soseščine. Algoritem A* je v povprečju optimiziral posamezne sklop dvajsetih opravil v 20 ms, algoritem soseščine pa v le 2 ms.



Slika 6.9: Časovna optimizacija – trajanje

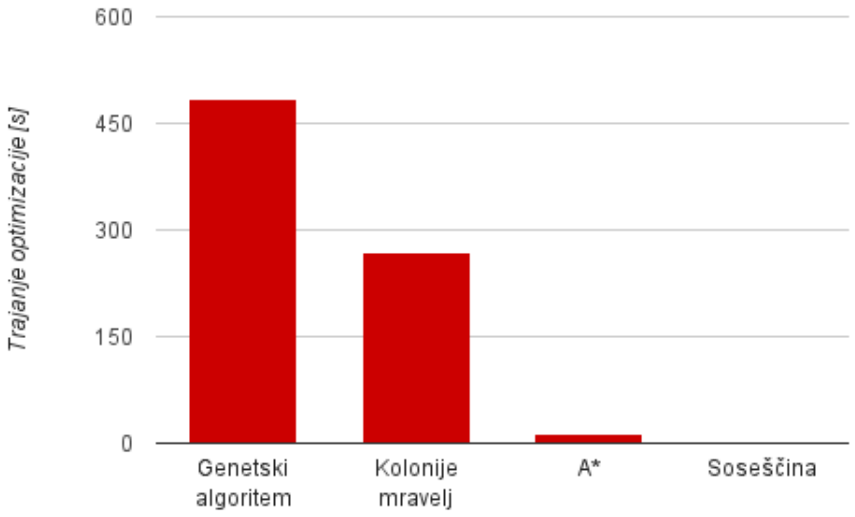
Optimizacijski rezultati časovne optimizacije, dinamičnega izvajanja opravil so prikazani na sliki 6.10. Genetskemu algoritmu in optimizaciji s kolonijami mravelj je uspelo časovno optimizirati izvedbo opravil za 11%. Oba algoritma sta imela povprečno starost opravil okoli 20, največjo starost opravila pa 33. Algoritem A* je dosegel 20,3 % pohitritev, algoritem soseščine pa celo 22,3 % pohitritev v primerjavi s sekvenčnem izvajanjem opravil. Tudi pri teh dveh algoritmih je bila povprečna starost opravil okoli 20, najvišja pa okoli 50.



Slika 6.10: Časovna optimizacija – optimiziranje

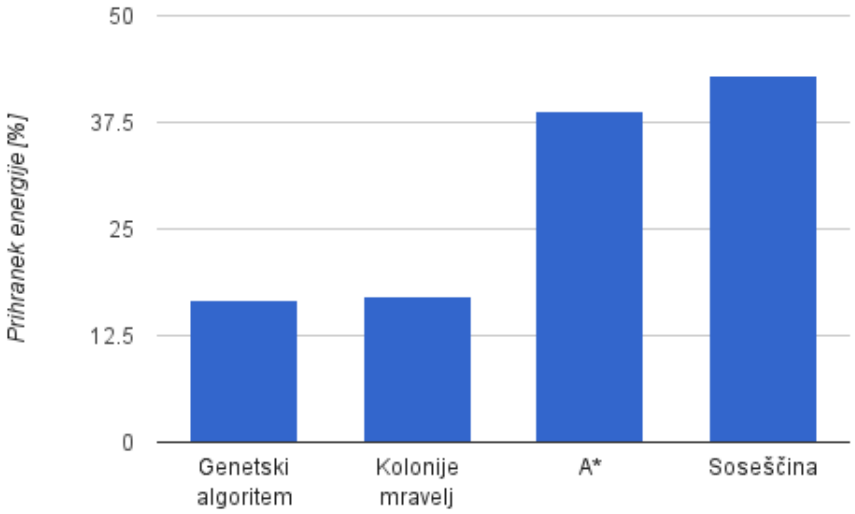
Energijska optimizacija

Pri energijski optimizaciji (slika 6.11), so bili časi posameznih algoritmov zopet zelo podobni časom časovne optimizacije. Največ časa za optimizacijo sta potrebovala genetski algoritem in optimizacija s kolonijami mravelj. Daleč v ospredju pa sta bila algoritem A* s časom 12,7 sekund in algoritem soseščine s časom 1,22 sekunde.



Slika 6.11: Energijska optimizacija – trajanje

Porabo energije pri energijski optimizaciji, dinamične izvedbe opravil (slika 6.12), je genetskemu algoritmu ter optimizaciji s kolonijami mravelj uspelo optimizirati za 17 %. Algoritem A* je dosegel 38,76 % optimizacijo energije, še boljši od njega pa je bil algoritem soseščine s 43 % energijsko optimizacijo.



Slika 6.12: Energijska optimizacija – optimiziranje

6.2 Primerjava različnih konfiguracij skladiščnih sistemov

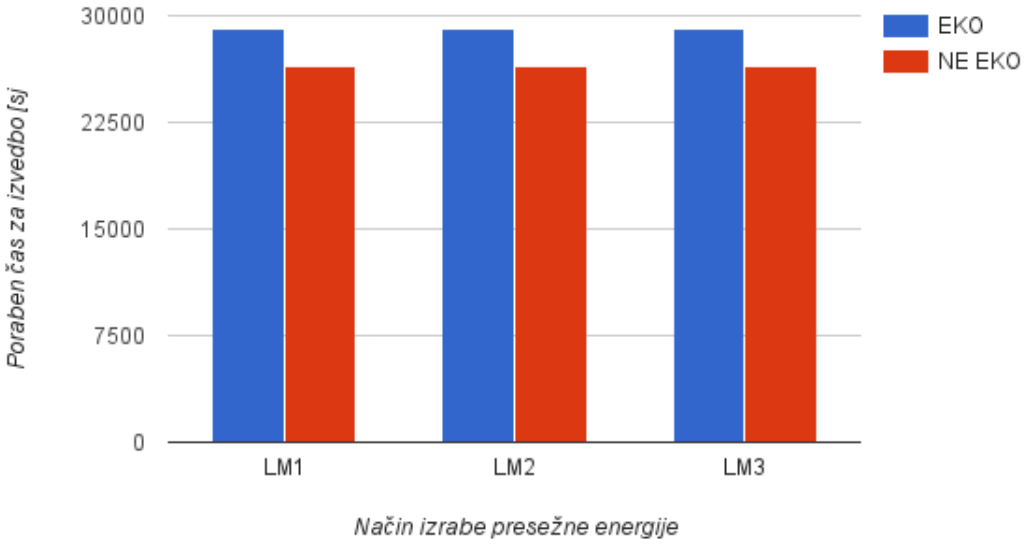
Analizo različnih konfiguracij skladiščnih sistemov smo opravljali nad algoritmom soseščine, ki se je na podlagi predhodnih analiz izkazal za najboljšega. Pri analiziranju sistema smo uporabili dinamično optimizacijo 20(19) petstotih umetno ustvarjenih opravil. Pri analiziranju smo spreminjali naslednje parametre skladiščnega sistema:

- Število miz na dvigalu:
  - 1, dvigalo z le eno mizo za transportno skladiščne enote.
  - 2, dvigalo z dvema mizama za transportno skladiščne enote.
- Premikanje dvigala:
  - EKO, predstavlja ekonomično premikanje dvigala. Pri spuščanju se x in y osi začneta premikati hkrati, pri dviganju pa se y os začne premikati takrat, ko x os začne zavirati. Tako je ekonomično premikanje počasnejše od ne ekonomičnega.
  - NE EKO, predstavlja neekonomično premikanje dvigala. Pri tem se x in y osi vedno začneta premikati hkrati.
- Način izrabe presežne energije med pogoni:
  - LM1, odvečna energija gre preko uporov.
  - LM2, pogoni si izmenjujejo energijo med sabo.
  - LM3, odvečna energija pogonov se vrača v omrežje.
- Verjetnost dvojnih opravil:
  - 0, generator ustvarja dvojna opravila z verjetnostjo 0 %.
  - 50, generator ustvarja dvojna opravila z verjetnostjo 50 %.
  - 100, generator ustvarja dvojna opravila z verjetnostjo 100 %.

6.2.1 Sekvenčna izvedba opravil

Posamezne čase izvedbe danih opravil in porabljeno energijo pri tem od posameznih algoritmov smo primerjali s sekvenčno razporeditvijo opravil. Sekvenčna izvedba, s katero smo primerjali rezultate optimizacije, je bila izračunana glede na takratne specifikacije skladiščnega sistema. Torej, pri optimizaciji skladiščnega sistema z izmenjavo energije preko uporov so se rezultati optimizacijski algoritmov primerjali s sekvenčno izvedbo opravil v skladiščnem sistemu, ki uporablja upore za izmenjavo energije. Tako smo lahko analizirali, koliko lahko dejansko pripomoremo k optimizaciji skladiščnih sistemov z našim optimizacijskimi algoritmi. Vendar pa nam takšna analiza ne pokaže dobro prednosti posameznih skladiščnih sistemov pred ostalimi, zato smo posebej analizirali čase ter energije sekvenčne izvedbe opravil pri posameznih konfiguracijah skladiščnega sistema.

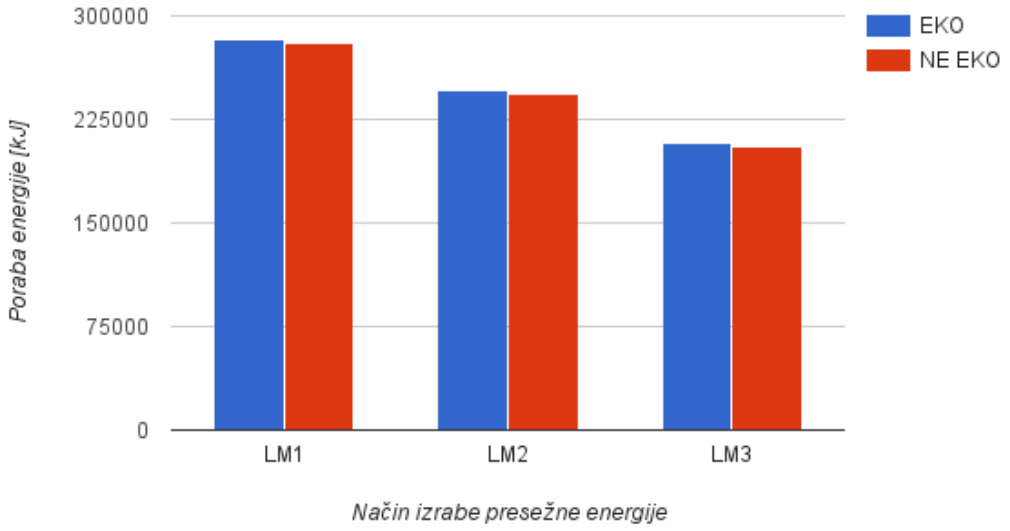
Časi pri sekvenčni izvedbi opravil (slika 6.13), so bili povsem enaki, ne glede na način izrabe presežne energije med pogoni. So pa bili časi pri neekonomičnem premikanju dvigala za 8,7 % manjši od časov pri ekonomičnem premikanju dvigala.



Slika 6.13: Potreben čas za sekvenčno izvedbo opravil

Pri porabi energije (slika 6.14), so se posamezne sekvenčne izvedbe opravil veliko bolj razlikovale kot pri porabi časa. Največ energije se je uporabilo pri prvem načinu izrabe presežne energije med pogoni $LM1$, kjer gre odvečna energija preko uporov stran. Za 12,9 % je bila manjša poraba energije pri drugem načinu $LM2$, kjer so si pogoni energijo med sabo izmenjevali. Najmanj energije pa smo porabili pri tretjem načinu $LM3$, kjer se odvečna energija vrača v omrežje. Pri tem načinu je bila za 26,2 % manjša poraba energije v primerjavi z prvim načinom izrabe presežne energije preko uporov.

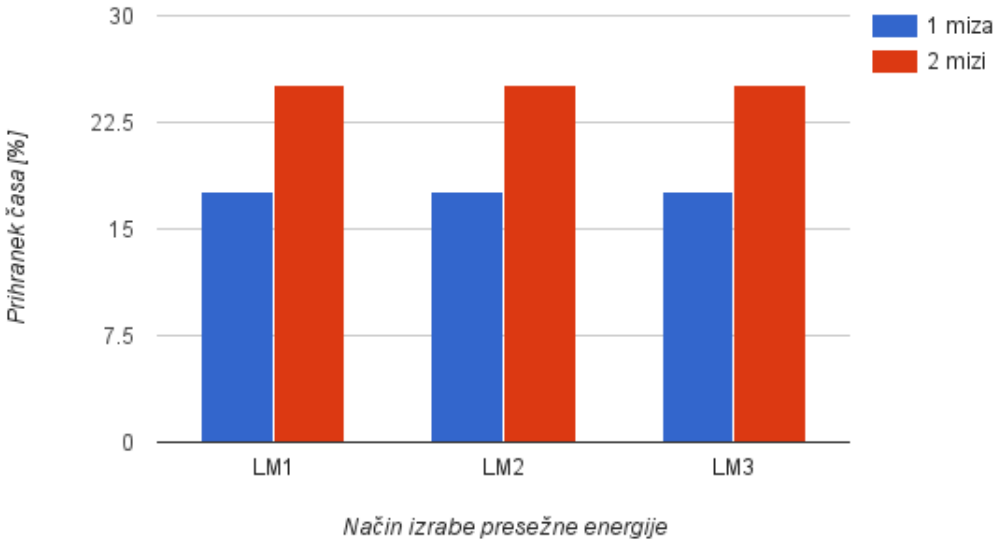
Med ekonomičnim in neekonomičnim premikanjem dvigala v porabi energije ni bilo velike razlike. Pri ekonomičnem premikanju se je v povprečju porabilo 1 % več energije kot pa pri neekonomičnem premikanju dvigala. Pričakovali smo večjo razliko v prid ekonomičnemu premikanju dvigala, še posebej pri drugem načinu izrabe presežne energije med pogoni. Večjo porabe energije pri ekonomične premikanju dvigala si razlagamo z daljšim časom izvajanja opravil, ki je bil posledica ekonomičnega premikanja dvigala. Poleg daljšega časa pa obstaja možnost, da bila naša ocena porabe ostalih naprav P_{0aux} previsoka.



Slika 6.14: Poraba energije pri sekvenčni izvedbi opravil

6.2.2 Časovna optimizacija

V primeru konfiguracije skladiščnega sistema, ko se dvigalo premika ekonomično (slika 6.15), razlik časovne optimizacije med različnimi načini izrabe presežne energije med pogoni ni bilo. Pri uporabi le ene mize dvigala smo optimizirali čas izvedbe pri vseh treh načinih izrabe presežne energije za 17,62 %, medtem ko smo pri uporabi dveh miz dvigala čas uspeli izboljšati za 25,1 %.



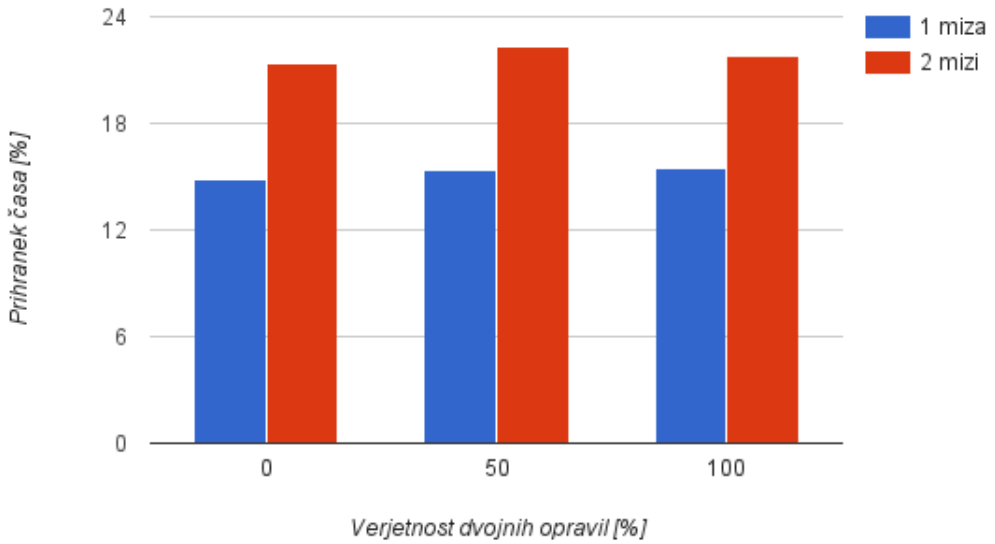
Slika 6.15: Časovna optimizacija – EKO

Ravno tako ni bilo razlik v doseženi časovni optimizaciji glede na način izrabe presežne energije med pogoni v primeru neekonomičnega premikanja dvigala. Z uporabo le ene mize smo dosegli 15,42 %, pri uporabi dveh miz na dvigalu pa 22,3 % optimizacijo razporeditve opravil.

Ne glede na izbrani način izrabe presežne energije med pogoni in število uporabljenih miz na dvigalu smo v povprečju dosegli za 2 % slabše časovne optimizacije pri neekonomičnem premikanju dvigala v primerjavi z ekonomičnem premikanjem. Izboljšanje rezultatov pri ekonomičnem premikanju dvigala smo tudi pričakovali, saj so zaradi daljšega časa potovanja dvigala optimi-

zacijski algoritmi imeli večji delež časa, ki so ga lahko optimizirali. Časa, ki ga dvigalo potrebuje za izvajanje akcij pobiranja ter razlaganja transportne skladiščne enote, algoritmi ne morejo optimizirati, razen v primeru dvojnih opravil. Medtem pa lahko v vsakem koraku optimizirajo čas, ki je potreben, da pripelje dvigalo transportno skladiščne enote z ene lokacije na drugo.

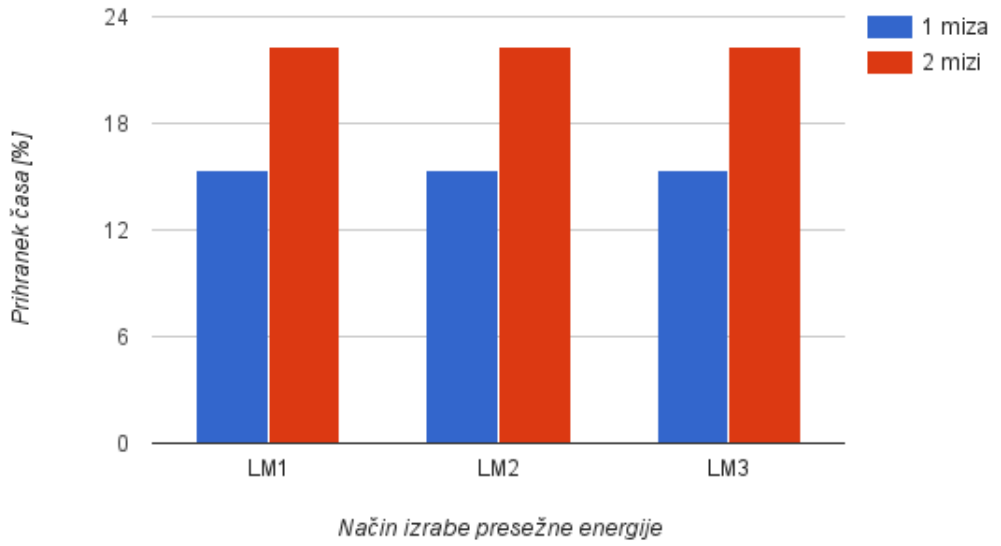
Pri optimizaciji opravil v odvisnosti od deleža dvojnih opravil (slika 6.16), so bile medsebojne razlike zelo majhne. Pri uporabi ene mize dvigala smo dosegli 15 % optimizacijo, pri uporabi dveh miz pa 22 %. Razlog za podobne rezultate in ne občutno izboljšanje pri večjem deležu dvojnih opravil so bila dvojna opravila, ki smo jih ustvarili. Ta so bila namreč le delno dvojna: dve opravili je mogoče hkrati začeti ali zaključiti ali pa enega začeti in drugega zaključiti. Za ustvarjanje le delnih dvojnih opravil smo se odločili zaradi našega skladiščnega sistema, ki ima le en par sosednjih V/I mest. V primeru pogoste uporabe istih V/I mest se opravila med sabo namreč blokirajo, s tem pa se nam zelo zmanjša optimizacijski prostor. To je razvidno tudi pri opravilih, kjer je delež dvojnih opravil 100 %, ko se zaradi blokiranja opravil optimizacija rahlo poslabša.



Slika 6.16: Časovna optimizacija v odvisnosti od deleža dvojnih opravil

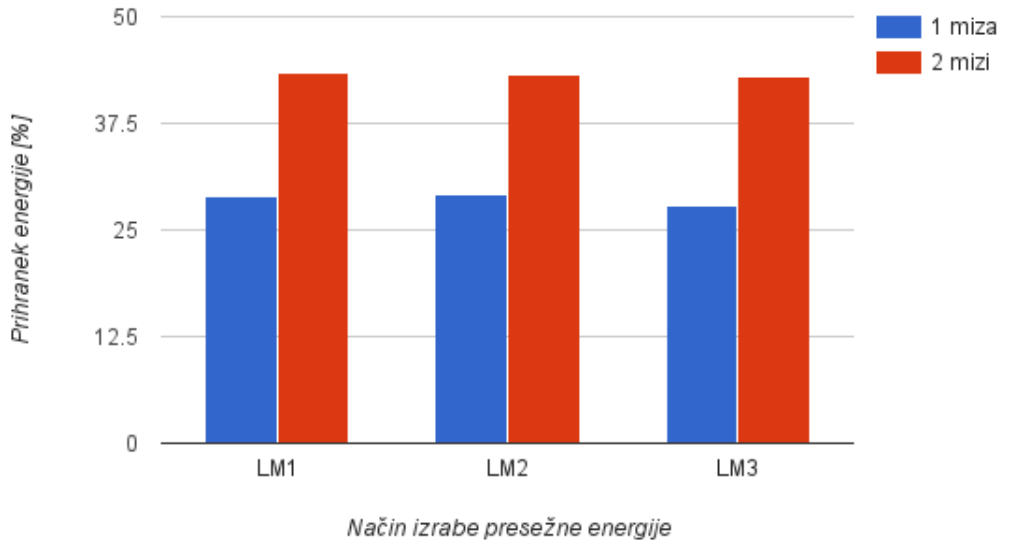
6.2.3 Energijska optimizacija

Pri optimizaciji energije v odvisnosti od načina izrabe presežne energije med pogoni so bile razlike med doseženimi optimizacijami minimalne (slika 6.17). Pri konfiguraciji skladiščnega sistema, kjer se dvigalo premika ekonomično, smo z uporabo ene mize v povprečju dosegli 28 % optimizacijo energije, z uporabo dveh miz pa kar 43 %.



Slika 6.17: Energijska optimizacija – EKO

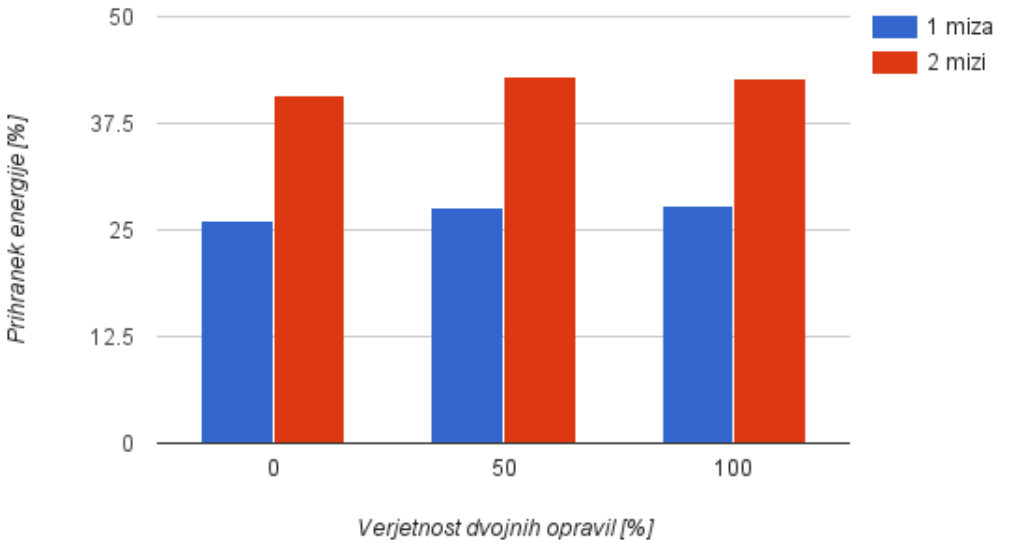
Ravno tako so bile zelo majhne razlike pri optimizaciji energije v odvisnosti od načina izrabe presežne energije med pogoni pri premikanju dvigala po neekonomičnem načinu (slika 6.18). Pri uporabi ene mize na dvigalu za transport transportno skladiščnih enot smo dosegli 27 % optimizacijo, pri uporabi dveh miz pa 42 % optimizacijo energije.



Slika 6.18: Energijska optimizacija – NE EKO

Tako smo dosegli pri ekonomičnem premikanju dvigala v povprečju za 1 % boljše optimizacijske rezultate energije kot pri neekonomičnem premikanju. Boljši rezultati pri ekonomičnem premikanju so bili pričakovani, vendar smo pričakovali večje razlike. Razloga za majhne razlike smo navedli že pri statični analizi opravil: daljši čas izvajanja opravil pri ekonomičnem premikanju dvigala in ocena porabe energije ostalih porabnikov v skladiščnem sistemu.

Pri optimizaciji opravil v odvisnosti od deleža dvojnih opravil so bile tudi pri energijski optimizaciji medsebojne razlike zelo majhne (slika 6.19). Pri uporabi ene mize dvigala smo dosegli 27 % optimizacijo, pri uporabi dveh miz pa 42 %. Razlog je bil zopet enak kakor pri časovni optimizaciji, torej blokiranje opravil med seboj zaradi strukture našega skladiščnega sistema.

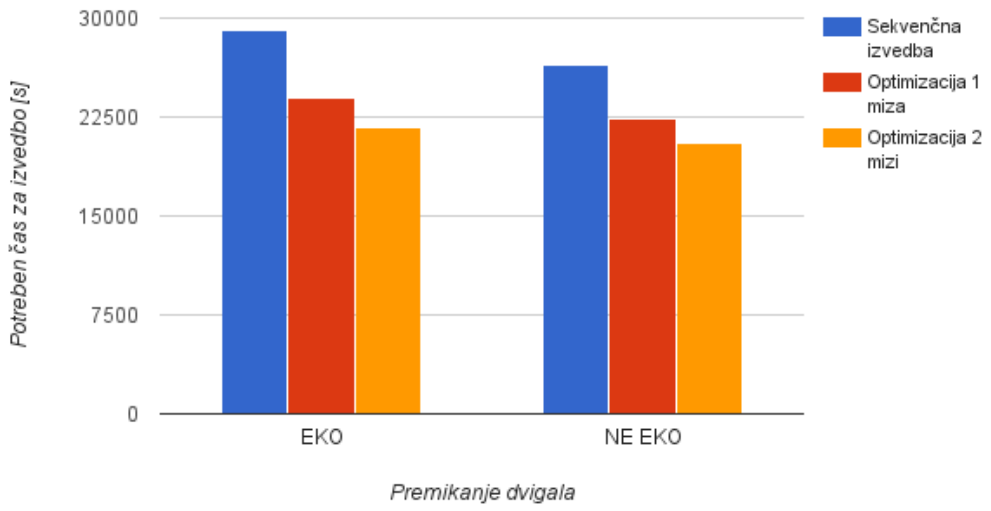


Slika 6.19: Energijska optimizacija v odvisnosti od deleža dvojnih opravil

6.2.4 Pregled različnih režimov

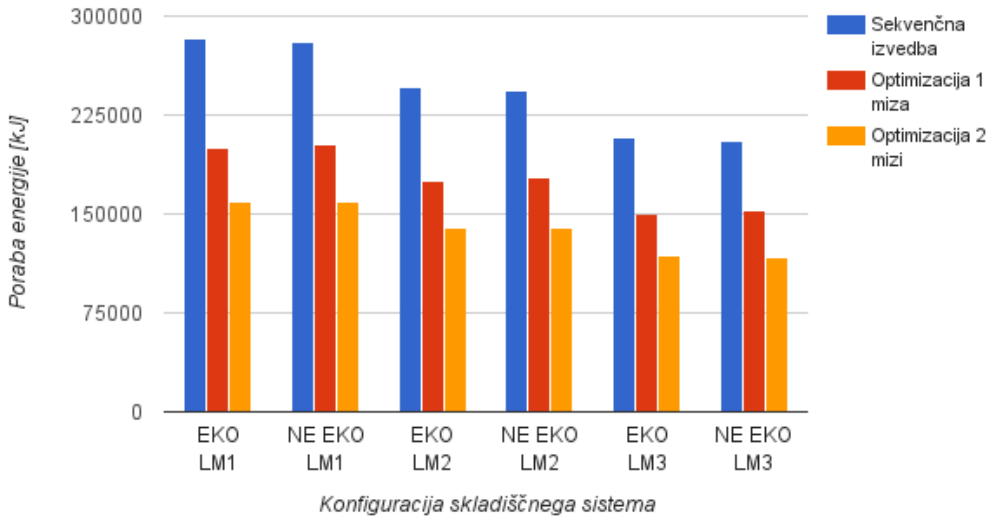
Za boljšo primerjavo različnih skladiščnih sistemov pred optimizacijo in po njej smo opravili še analizo absolutnih rezultatov sekvenčne izvedbe in optimizirane izvedbe opravil.

Potrebni časi za izvedbo opravil pri različnih konfiguracijah skladiščnega sistema so prikazani na sliki 6.20. Na sliki so tako časi sekvenčne izvedbe opravil brez optimizacije, časi optimizirane izvedbe opravil z uporabo le ene mize na dvigalu in časi optimizirane izvedbe opravil pri uporabi dveh miz na dvigalu. Pri različnih konfiguracijah skladiščnega sistema so prikazani le rezultati za različne načine premikanja dvigala, ne pa tudi rezultati za različne načine izrabe presežne energije. Razlog za to je, da rezultati pri časovni optimizaciji niso odvisni od načina izrabe presežne energije. Na sliki opazimo, da je ekonomičen način premikanja dvigala počasnejši, kar smo tudi pričakovali. V obeh primerih pa z optimizacijo zelo zmanjšamo čas izvedbe opravil, tako pri uporabi ene mize dvigala kakor pri uporabi dveh miz.



Slika 6.20: Čas izvedbe opravil pri različnih konfiguracijah skladiščnega sistema

Porabljena energija za izvedbo opravil pri različnih konfiguracijah skladiščnega sistema je prikazana na sliki 6.21. Na sliki so tako energije sekvenčne izvedbe opravil brez optimizacije, energije optimizirane izvedbe opravil z uporabo le ene mize na dvigalu in energije optimizirane izvedbe opravil pri uporabi dveh miz na dvigalu. Razlika pri porabi energije med ekonomičnim premikanjem in neekonomičnim premikanjem dvigala je v vseh primerih minimalna. Pričakovali smo večjo razliko v prid ekonomičnega premikanja pri načinu izrabe presežne energije LM1 in LM2. Po pričakovanjih pa se je energija zelo razlikovala pri različnih načinih izrabe presežne energije med pogoni tako pri neoptimizirani izvedbi kakor pri optimizirani izvedbi opravil. Potrebna energija pri optimizirani izvedbi opravil z uporabo načina izrabe odvečne energije LM3 je za skoraj 60 % manjša kot pri neoptimizirani izvedbi opravil pri uporabi načina izrabe presežne energije LM1.



Slika 6.21: Poraba energije pri izvedbi opravil pri različnih konfiguracijah skladiščnega sistema

Poglavje 7

Sklepne ugotovitve

Cilj magistrskega dela je bila časovna ter energijska optimizacija izvedbe opravil v visokoregalnem skladiščnem sistemu z napravo, ki omogoča prenos več transportno skladiščnih enot hkrati. Zahteva, ki smo jo imeli pri optimizaciji, je bila, da poteka v realnem času, torej ima posamezni algoritem za optimizacijo na voljo le nekaj sekund. Po sami optimizaciji skladiščnega sistema pa smo želeli izvesti še primerjavo takšnega skladiščnega sistema s skladiščnim sistemom, ki uporablja napravo z le eno mizo za transportno skladiščne enote.

Problema smo se lotili tako, da smo najprej implementirali osnovne algoritme za reševanje takšnega problema. Algoritmi, ki smo jih implementirali, so bili Dijkstrov algoritem, Bellman-Fordov algoritem, genetski algoritem, optimizacija s kolonijami mravelj, algoritem A* in algoritem soseščine. Algoritem soseščine je algoritem, ki smo si ga zamislili sami in deluje po sistemu odločanja in ne po preiskovalnem načinu kakor ostali omenjeni algoritmi. Po sami implementaciji algoritmov smo izvedli optimizacijo izvajanja opravil skladiščnega sistema pri osnovnih konfiguracijah.

Najboljše optimizacijske rezultate sta nam vračala Dijkstrov algoritem ter Bellman-Fordov algoritem, kakor smo predvidevali. Vendar pa sta ta dva algoritma presegla naše časovne zahteve že pri optimizaciji štirih opravil skladiščnega sistema. Pri genetskemu algoritmu in pri optimizaciji s kolonijami mravelj smo dosegli dobre rezultate pri manjšem številu opravil za optimizacijo. Pri večjem številu opravil pa so se ti rezultati zelo poslabšali, ker je postal preiskovalni prostor za algoritma prevelik. V primeru, ko smo spremenili določene parametre algoritmov, smo uspeli doseči dobre rezultate tudi pri večjem številu opravil, vendar pa smo pri tem prekoračili zadane časovne omejitve. V splošnem, ne glede na število opravil, smo najboljše rezultate dosegli z algoritmom A* in algoritmom soseščine. Algoritem soseščine je bil po optimizacijskih rezultatih in času izvajanja rahlo v prednosti pred algoritmom A*. Algoritem A* je uspel 500 iteracij po 20 opravil optimizirati v desetih sekundah, pri čemer je bil njegov povprečni čas optimiziranja ene iteracije 20 ms. Algoritem soseščine pa je uspel vse optimizirati v eni sekundi s povprečnim časom 2 ms na iteracijo. Tako nam je z obema algoritmoma uspelo optimizirati čas izvedbe opravil za 20 %, porabo energije pa za kar 40 %.

Pri analiziranju optimizacij pri različnih konfiguracijah skladiščnega sistema z algoritmom soseščine smo spreminjali način premikanja dvigala, število miz na dvigalu ter način izrabe presežne energije. Imeli smo dva načina premikanja, ekonomično in neekonomično. Pri ekonomičnem premikanju dvigala se pri spuščanju x in y osi začneta premikati hkrati, pri dviganju pa se y os začne premikati takrat, ko x os začne zavirati. Pri neekonomičnem premikanju pa se x in y osi vedno začneta premikati hkrati, tako je neekonomično premikanje dvigala hitrejše od ekonomičnega. Optimizacijski rezultati so pokazali, da se v našem primeru ekonomično premikanje dvigala ne obrestuje niti časovno niti energijsko. Za naš skladiščni sistem je tako bolj primerno neekonomično premikanje dvigala. Drugi parameter, ki smo ga spreminjali, je bilo število miz na dvigalu za prenos transportno skladiščnih enot. Ta

parameter smo preklapljali med eno mizo ter dvema mizama. Pri eni mizi smo dosegli 15 % časovne ter 27 % energijske optimizacije v primerjavi s sekvenčno izvedbo opravil v zaporedju, kakršnem jih ustvari sistem WMS. Pri uporabi dveh miz pa smo dosegli pri časovni optimizaciji 22 % pohitritve, pri energiji pa smo porabo zmanjšali za kar 43 %. V našem skladiščnem sistemu z dvema mizama na dvigalu torej dosežemo večji izkoristek kot v sistemu z eno mizo, kar smo tudi pričakovali. Pri spreminjanju načina izrabe presežne energije smo analizirali primer, ko gre odvečna energija preko uporov, ki se izmenjuje med pogoni, ter primer, ko se vrača v omrežje. Pri obeh načinih smo z uporabo dveh miz na dvigalu v primerjavi s sekvenčno izvedbo zmanjšali energijo za okoli 43 %. Vendar pa so se sekvenčne izvedbe razlikovale glede na posamezni način izrabe presežne energije med pogoni. Pri izmenjavi energije med pogoni je bila že sekvenčna izvedba opravil energijsko za 13 % varčnejša od načina, kjer gre energija preprosto preko uporov. Ravno tako je bila pri načinu, kjer se odvečna energija vrača v omrežje, sekvenčna izvedba opravil za 15 % varčnejša od načina, kjer se energija izmenjuje med pogoni. Energijsko se tako najbolj izplača implementirati način izrabe presežne energije med pogoni, kjer se odvečna energija vrača v omrežje.

V nadaljevanju bi bilo dobro izvesti optimizacijo še na drugih skladiščnih sistemih z drugačno razporeditvijo V/I mest. Namreč velika pomanjkljivost našega skladiščnega sistema pri optimizaciji, je bila ta, da je imel le en par sosednjih V/I mest. Ravno tako bi bilo zanimivo opraviti analizo za primer skladiščnega sistema, kjer s transportno skladiščnimi enotami upravlja več naprav. Pri tem bi bilo potrebno našemu algoritmu soseščine dodelati logiko, ki bi podpirala delo z več napravami osnovno ogrodje bi pa lahko ostalo enako. V primeru optimizacije z uporabe algoritma A* pa bi bila potrebno spremeniti hevristično funkcijo. Predvidevamo, da bi bili dobljeni rezultati veliko boljši od naših, še posebej v primeru uporabe skladiščnega sistema, ki ima drugačno razporeditev V/I mest.

Literatura

[1] L. Jian, W. Xin, W. Weize, L. Changlong, Z. Ting, Z. Yang, Route optimization based on genetic algorithms of stacker for automated storage and retrieval system, in: Intelligent Systems (GCIS), 2010 Second WRI Global Congress on, Vol. 1, 2010, pp. 243–248. doi:10.1109/GCIS.2010.11.

[2] H. Ma, S. Su, D. Simon, M. Fei, Ensemble multi-objective biogeography-based optimization with application to automated warehouse scheduling, Engineering Applications of Artificial Intelligence 44 (2015) 79 – 90. doi:http://dx.doi.org/10.1016/j.engappai.2015.05.009.
URL http://www.sciencedirect.com/science/article/pii/S0952197615001207

[3] H.-P. Li, Z.-F. Fang, S.-Y. Ji, Research on the slotting optimization of automated stereoscopic warehouse based on discrete particle swarm optimization, in: Industrial Engineering and Engineering Management (IE EM), 2010 IEEE 17Th International Conference on, 2010, pp. 1404–1407. doi:10.1109/ICIEEM.2010.5646028.

[4] V. Colla, G. Nastasi, N. Matarese, L. M. Reyneri, Ga-based solutions comparison for storage strategies optimization for an automated warehouse, in: Intelligent Systems Design and Applications, 2009. ISDA'09. Ninth International Conference on, IEEE, 2009, pp. 976–981.

[5] Mecalux, Mecalux, Warehouse solutions Warehouse solutions, http://www.mecalux.com/automated-warehouses-for-pallets/conveyor-system-for-pallets, [Splet; dostopno 09.06.2016] (2016).

[6] M. Hompel, T. Schmidt, Warehouse management: automation and organisation of warehouse and order picking systems, Springer Science & Business Media, 2006.

[7] U. Dharmapriya, A. Kulatunga, New strategy for warehouse optimization–lean warehousing, in: Proceedings of the 2011 International Conference on Industrial Engineering and Operations Management, 2011, pp. 513–519.

[8] J. Karásek, High-Level Object. Oriented Genetic Programming in Logistic Warehouse Optimization, Ph.D. thesis, Faculty of Electrical Engineering and Communication department of telecommunications (2014).

[9] R. de Koster, T. Le-Duc, K. J. Roodbergen, Design and control of warehouse order picking: A literature review, European Journal of Operational Research 182 (2) (2007) 481 – 501. doi:http://dx.doi.org/10.1016/j.ejor.2006.07.009.
URL http://www.sciencedirect.com/science/article/pii/S0377221706006473

[10] J. C.-H. Pan, P.-H. Shih, M.-H. Wu, J.-H. Lin, A storage assignment heuristic method based on genetic algorithm for a pick-and-pass warehousing system, Computers Industrial Engineering 81 (2015) 1 – 13. doi:http://dx.doi.org/10.1016/j.cie.2014.12.010.
URL http://www.sciencedirect.com/science/article/pii/S0360835214004392

[11] H. J. Carlo, G. E. Giraldo, Toward perpetually organized unit-load warehouses, Computers Industrial Engineering 63 (4) (2012) 1003 – 1012. doi:http://dx.doi.org/10.1016/j.cie.2012.06.012.

URL http://www.sciencedirect.com/science/article/pii/S0360835212001611

[12] G. N. Maschietto, Y. Ouazene, F. Yalaoui, M. C. de Souza, M. G. Ravetti, Two formulations for non-interference parallel machine scheduling problems, IFAC-PapersOnLine 48 (3) (2015) 272 – 276, 15th IFAC Symposium on Information Control Problems in Manufacturing (INCOM 2015). doi:http://dx.doi.org/10.1016/j.ifacol.2015.06.093.
URL http://www.sciencedirect.com/science/article/pii/S2405896315003328

[13] R. Dornberger, T. Hanne, R. Ryter, M. Stauffer, Optimization of the picking sequence of an automated storage and retrieval system (AS/RS), in: Evolutionary Computation (CEC), 2014 IEEE Congress on, 2014, pp. 2817–2824. doi:10.1109/CEC.2014.6900254.

[14] C. Theys, O. Bräysy, W. Dullaert, B. Raa, Using a TSP heuristic for routing order pickers in warehouses, European Journal of Operational Research 200 (3) (2010) 755 – 763. doi:http://dx.doi.org/10.1016/j.ejor.2009.01.036.
URL http://www.sciencedirect.com/science/article/pii/S0377221709000514

[15] R. K. Ahuja, K. Mehlhorn, J. Orlin, R. E. Tarjan, Faster algorithms for the shortest path problem, J. ACM 37 (2) (1990) 213–223. doi:10.1145/77600.77615.
URL http://doi.acm.org/10.1145/77600.77615

[16] M. Thorup, Undirected single-source shortest paths with positive integer weights in linear time, J. ACM 46 (3) (1999) 362–394. doi:10.1145/316542.316548.
URL http://doi.acm.org/10.1145/316542.316548

[17] S. Alexander, On the history of combinatorial optimization (till 1960), Handbooks in Operations Research and Management Science: Discrete Optimization 12 (2005) 1.

[18] J. Bang-Jensen, G. Z. Gutin, Digraphs: theory, algorithms and applications, Springer Science & Business Media, 2008.

[19] J. Y. Yen, An algorithm for finding shortest routes from all source nodes to a given destination in general networks, Quarterly of Applied Mathematics (1970) 526–530.

[20] M. Mitchell, An introduction to genetic algorithms, MIT press, 1998.

[21] J. Beasley, P. Chu, A genetic algorithm for the set covering problem, European Journal of Operational Research 94 (2) (1996) 392 – 404. doi:http://dx.doi.org/10.1016/0377-2217(95)00159-X.
URL http://www.sciencedirect.com/science/article/pii/037722179500159X

[22] D. Whitley, A genetic algorithm tutorial, Statistics and Computing 4 (2) (1994) 65–85. doi:10.1007/BF00175354.
URL http://dx.doi.org/10.1007/BF00175354

[23] C. W. Ahn, R. S. Ramakrishna, A genetic algorithm for shortest path routing problem and the sizing of populations, IEEE Transactions on Evolutionary Computation 6 (6) (2002) 566–579. doi:10.1109/TEVC.2002.804323.

[24] Y. Kaya, M. Uyar, et al., A novel crossover operator for genetic algorithms: ring crossover, arXiv preprint arXiv:1105.0355.

[25] J. Magalhaes-Mendes, A comparative study of crossover operators for genetic algorithms to solve the job shop scheduling problem, WSEAS Trans. Comput 12 (4) (2013) 164–173.

[26] M. Dorigo, L. M. Gambardella, Ant colonies for the travelling salesman problem, Biosystems 43 (2) (1997) 73 – 81. doi:http://dx.doi.org/10.1016/S0303-2647(97)01708-5.
URL http://www.sciencedirect.com/science/article/pii/S0303264797017085

[27] A. Colorni, M. Dorigo, V. Maniezzo, et al., Distributed optimization by ant colonies, in: Proceedings of the first European conference on artificial life, Vol. 142, Paris, France, 1991, pp. 134–142.

[28] I. Pešl, V. Žumer, J. Brest, Optimizacija s pomočjo kolonije mravelj, Elektrotehniški vestnik 73 (2/3) (2006) 93–98.

[29] M. Dorigo, C. Blum, Ant colony optimization theory: A survey, Theoretical Computer Science 344 (2–3) (2005) 243 – 278. doi:http://dx.doi.org/10.1016/j.tcs.2005.05.020.
URL http://www.sciencedirect.com/science/article/pii/S0304397505003798

[30] M. Dorigo, T. Stützle, Ant colony optimization: overview and recent advances, Techreport, IRIDIA, Universite Libre de Bruxelles.

[31] M. Dorigo, M. Birattari, T. Stutzle, Ant colony optimization, IEEE Computational Intelligence Magazine 1 (4) (2006) 28–39. doi:10.1109/MCI.2006.329691.

[32] M. Dorigo, L. M. Gambardella, Ant colony system: a cooperative learning approach to the traveling salesman problem, IEEE Transactions on Evolutionary Computation 1 (1) (1997) 53–66. doi:10.1109/4235.585892.

[33] L. H. O. Rios, L. Chaimowicz, A survey and classification of A* based best-first heuristic search algorithms, in: Advances in Artificial Intelligence–SBIA 2010, Springer, 2010, pp. 253–262.

[34] P. E. Hart, N. J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Transactions on Systems Science and Cybernetics 4 (2) (1968) 100–107. doi:10.1109/TSSC.1968.300136.

[35] F. Duchoň, A. Babinec, M. Kajan, P. Beňo, M. Florek, T. Fico, L. Jurišica, Path planning with modified a star algorithm for a mobile robot, Procedia Engineering 96 (2014) 59 – 69, modelling of Mechanical and Mechatronic Systems. doi:http://dx.doi.org/10.1016/j.proeng.2014.12.098.
URL http://www.sciencedirect.com/science/article/pii/S187770581403149X

[36] R. Dechter, J. Pearl, Generalized best-first search strategies and the optimality of A*, J. ACM 32 (3) (1985) 505–536. doi:10.1145/3828.3830.
URL http://doi.acm.org/10.1145/3828.3830

[37] A. Patel, Introduction to A*, http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html, [Splet; dostopno 01.06.2016] (2016).

[38] J. Yao, C. Lin, X. Xie, A. J. Wang, C. C. Hung, Path planning for virtual human motion using improved A* star algorithm, in: Information Technology: New Generations (ITNG), 2010 Seventh International Conference on, 2010, pp. 1154–1158. doi:10.1109/ITNG.2010.53.

[39] M. Greenspan, M. Yurick, Approximate k-d tree search for efficient icp, in: 3-D Digital Imaging and Modeling, 2003. 3DIM 2003. Proceedings. Fourth International Conference on, 2003, pp. 442–448. doi:10.1109/IM.2003.1240280.

[40] J. L. Bentley, Multidimensional binary search trees used for associative searching, Commun. ACM 18 (9) (1975) 509–517. doi:10.1145/

361002.361007.
URL http://doi.acm.org/10.1145/361002.361007

[41] A. W. Moore, An intoductory tutorial on KD-trees.